View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Calhoun, Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

Theses and Dissertations | Thesis Collection

2015-06

Discovery and optimization of low-storage Runge-Kutta methods

Fletcher, Matthew T.

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/45852

http://www.nps.edu/library

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DISCOVERY AND OPTIMIZATION OF LOW-STORAGE RUNGE-KUTTA METHODS

by

Matthew T. Fletcher

June 2015

Thesis Co-Advisors: Lucas C. Wilcox
Jeremy E. Kozdon

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	06-12-2015	Master's Thesis 06-30-2013 to 06-12-2015

4. TITLE AND SUBTITLE

DISCOVERY AND OPTIMIZATION OF LOW-STORAGE RUNGE-KUTTA METHODS

5. FUNDING NUMBERS

6. AUTHOR(S)

Matthew T. Fletcher

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Runge-Kutta (RK) methods are an important family of iterative methods for approximating the solutions of ordinary differential equations (ODEs) and differential algebraic equations (DAEs). It is common to use an RK method to discretize in time when solving time dependent partial differential equations (PDEs) with a method-of-lines approach as in Maxwell's Equations. Different types of PDEs are discretized in such a way that could result in a high dimensional ODE or DAE. We use a low-storage RK (LSRK) method that stores two registers per ODE dimension, which limits the impact of increased storage requirements. Classical RK methods, however, have one storage variable per stage. In this thesis we compare the efficiency and accuracy of LSRK methods to RK methods. We then focus on optimizing the truncation error coefficients for LSRK to discover new methods. Reusing the tools from the optimization method, we discover new methods for low-storage half-explicit RK (LSHERK) methods for solving DAEs.

14. SUBJECT TERMS	15. NUMBER OF PAGES
low-storage Runge-Kutta (LSRK),stability region, Maxwell's equations, half-explicit Runge-Kutta (HERK), low-storage half-explicit runge-kutta (LSHERK), differential algebraic equation (DAE)	101
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DISCOVERY AND OPTIMIZATION OF LOW-STORAGE RUNGE-KUTTA METHODS

Matthew T. Fletcher
Captain, United States Army
B.S., United States Military Academy, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June 2015

Author: Matthew T. Fletcher

Approved by: Lucas C. Wilcox
Thesis Co-Advisor

Jeremy E. Kozdon
Thesis Co-Advisor

Craig W. Rasmussen
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Runge-Kutta (RK) methods are an important family of iterative methods for approximating the solutions of ordinary differential equations (ODEs) and differential algebraic equations (DAEs). It is common to use an RK method to discretize in time when solving time dependent partial differential equations (PDEs) with a method-of-lines approach as in Maxwell's Equations. Different types of PDEs are discretized in such a way that could result in a high dimensional ODE or DAE. We use a low-storage RK (LSRK) method that stores two registers per ODE dimension, which limits the impact of increased storage requirements. Classical RK methods, however, have one storage variable per stage. In this thesis we compare the efficiency and accuracy of LSRK methods to RK methods. We then focus on optimizing the truncation error coefficients for LSRK to discover new methods. Reusing the tools from the optimization method, we discover new methods for low-storage half-explicit RK (LSHERK) methods for solving DAEs.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

DG	discontinuous Galerkin
HERK	half-explicit Runge-Kutta
IVP	initial value problem
LSHERK	low-storage half-explicit Runge-Kutta
LSRK	low-storage Runge-Kutta
MOL	method-of-lines
ODE	ordinary differential equation
PDE	partial differential equation
RK	Runge-Kutta
TEC	truncation error coefficient

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to thank my advisor, Dr. Lucas Wilcox, who directed me through countless hours of coding and analysis. My understanding of these topics increased with each revision. His patience throughout the writing process and knowledge of these topics forced me to work hard. Furthermore, I would like to thank Dr. Jeremy Kozdon. He introduced me to numerical methods and established the base of my knowledge used throughout this thesis. I started graduate school with very little coding experience. His office became a second classroom for a few months while I learned how to code my first RK method. I also express my gratitude to Dr. Frank Giraldo. Sometimes it felt like I was working on a thesis during his Galerkin methods course, but I gained an understanding and respect for the work that goes into solving ODEs and PDEs. A special thanks goes out to my fellow Applied Mathematicians, Phil Baxa, Ben Davis and Russ Nelson. We each mastered different topics and helped one another through more than a few difficult courses. Finally, I would like to thank my beautiful wife, Dana. Without her I would not be where I am today. Her work ethic and positive attitude push me to be better every day.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Runge-Kutta (RK) methods are an important family of iterative methods for approximating the solutions of ordinary differential equations (ODEs) and differential algebraic equations (DAEs). ODEs involve differential equations whereas a DAE involves both differential equations and algebraic constraints. RK methods are single step methods that advance the ODE or DAE through a series of intermediate stage computations. For example, in the method of lines approach to discretizing time dependent partial differential equations (PDEs) like Maxwell's Equations, it is common to use RK methods to discretize in time. Depending on the PDE this can result in a large system of ODEs or DAEs. Storage requirements for standard RK methods are proportional to the number of stage computations. Low storage RK (LSRK) methods are a subclass of RK methods whose storage requirements are independent of the number of stage computations. This allows the increase of the number of stages without increasing storage expense. Some research proposed that we should increase the number of stages in order to increase the time step size thus speeding up time to solution for solving PDEs.

The goal of this thesis is to explore this idea for ODEs and to generate new LSRK methods for both ODEs and DAEs. In this work, we implement and test fourth order LSRK methods. We verify the accuracy conditions for the implemented methods. We also look at the efficiency of the LSRK methods along with the classic fourth order RK method using a simple ODE as well as a method of lines discretization of Maxwell's Equation in 2D. We explore the development of new LSRK schemes and attempt to optimizes them by minimizing the terms in local truncation error. We conclude this work by developing and testing half-explicit RK methods with a low storage format for solving DAEs.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Building a Low-Storage Runge-Kutta Method

This chapter focuses on the elements required to build and check an LSRK method. We introduce RK methods and what a low-storage implementation of RK means. We then show how to derive an accuracy condition while we list out the order conditions up to fifth order. We show how to construct a Butcher Tableau from an LSRK tableau of coefficients. Next, we bring the order conditions and the Butcher Tableau from an LSRK method together in order to ensure the LSRK method satisfies the accuracy conditions. We then compare methods that satisfy the same order conditions using the truncation error coefficient (TEC), a surrogate for local truncation error. The last section deals with the stability region of the RK method.

2.1 Runge-Kutta Methods

We consider the initial value problem (IVP)

$$y'(t) = f(t, y(t)), \quad y(t(0)) = y_0, \tag{2.1}$$

where f and y are vector functions. We want to find the numerical approximation of the solution $y(t)$ of the IVP over the time interval $t \in [t_0, t_f]$. We subdivide the interval $[t_0, t_f]$ into M equally spaced subintervals in which we integrate the solution; this choice of an equally spaced grid is not required for RK. Thus we have approximation points

$$h = \frac{t_f - t_0}{M}, \tag{2.2}$$

$$t_n = t_0 + nh, \quad n = 0, 1, \ldots, M, \tag{2.3}$$

where h is the time step size. We use an RK method to obtain an approximation of $y(t_n)$ using the solution value at $y(t_{n-1})$. One step of a general, explicit RK method

c_1	0			
$\vdots$	$a_{2,1}$	$\ddots$		
$\vdots$	$\vdots$	$\ddots$		
c_s	$a_{s,1}$	$\dots$	$a_{s,s-1}$	0
	b_1		$\dots$	b_s

Table 2.1: A general, explicit Butcher Tableau, from [1].

for numerically solving Equation (2.1) is

$$K_i = f\left(t_{n-1} + c_i h, y_{n-1} + h\sum_{j=1}^{i-1} a_{ij}K_j\right), \quad i = 1,...,s, \tag{2.4}$$

$$y_n = y_{n-1} + h\sum_{i=1}^{s} b_i K_i. \tag{2.5}$$

The variable s is the number of stages. The a_{ij} coefficients are the intermediate weights at each RK stage, b_j are the final stage weights, and c_i are the intermediate time levels. We require that

$$c_i = \sum_{j=1}^{s} a_{ij}, \tag{2.6}$$

since we want the RK integration of Equation (2.1) and its associated autonomous version to be discretely equivalent. The autonomous version of Equation (2.1) is

$$\hat{y}'(t) = \hat{f}(\hat{y}(t)), \tag{2.7}$$

where

$$\hat{y} = \begin{pmatrix} y \\ t \end{pmatrix}, \quad \hat{f} = \begin{pmatrix} f \\ 1 \end{pmatrix}. \tag{2.8}$$

One way to represent an explicit RK scheme with s stages is with a Butcher Tableau [1]. Table 2.1 shows the general, explicit Butcher Tableau for Equations (2.4) and (2.5), which encompass various RK methods. One of the most widely used RK methods is a fourth order, four stage method, referred to as RK4. The tableau for RK4 is given in Table 2.2 and Algorithm 2.1 gives a MATLAB implementation.

```
function [yn,t] = RK4(yn,f,t,h,M)
% yn = initial y
% f = anonymous function f = @(t,y)
% t = time
% h = time step size
% M = number of steps
for n = 1:M
    K1 = f(t,yn);
    K2 = f(t + h/2,yn + h/2*K1);
    K3 = f(t + h/2,yn + h/2*K2);
    K4 = f(t + h  ,yn + h  *K3);
    yn = yn + h/6 * (K1+2*K2+2*K3+K4);
    t  = t + h;
end
end
```

Algorithm 2.1: Method for RK4.

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

Table 2.2: The Butcher Tableau formulation of RK4.

In Algorithm 2.1 we evaluate the right-hand side function, f, four different times. We also store a total of five different variables, one for each iteration of RK4. LSRK methods are a specific class of RK methods, which require fewer storage registers. Williamson [2] demonstrated that some RK schemes can be implemented in a 2N-storage format, where N is the dimension of the ODE. This format requires only two registers of length N to implement. One step of the general s-stage 2N LSRK method is

$$S_1^{[1]} = y_n, \tag{2.9}$$

$$\left.\begin{aligned} S_2^{[i+1]} &= A_i S_2^{[i]} + h f(t_n + c_i h, S_1^{[i]}), \\ S_1^{[i+1]} &= S_1^{[i]} + B_i S_2^{[i]}, \end{aligned}\right\} \quad i = 1, \ldots, s, \tag{2.10}$$

$$y_{n+1} = S_1^{[s]}. \tag{2.11}$$

As shown in Algorithm 2.2, only two variables S_1 and S_2 are required to implement the LSRK method. Algorithm 2.2 gives an implementation of this LSRK method [3].

```
function [S1] = LSRK(f,y0,A,B,c,t,h)
% f = anonymous function f = @(t,y)
% y0 = initial condition
% A = A coefficients from the LSRK tableau
% B = B coefficients from the LSRK tableau
% c = c coefficients derived from the Butcher tableau
% t = time
% h = the time step size
s = length(A);
S1=y0;
S2=zeros(size(y0));
for i = 1:s
    S2 = A(i)*S2+h.*f(t+c(i)*h,S1')';
    S1 = S1+B(i)*S2;
end
end
```

Algorithm 2.2: Method for implementing LSRK methods.

Much like the Butcher Tableau, we display the coefficients of an LSRK scheme in two arrays

$$\begin{array}{c|c} A_1 & B_1 \\ \vdots & \vdots \\ A_s & B_s \end{array}.$$

We do not list the c_i terms since they come from Equation (2.6). In order to determine the LSRK coefficients, we define the relationship between the Butcher coefficients and the new LSRK coefficients with the equations

$$B_j = a_{j+1,j} \text{ when } j \neq M, \tag{2.12}$$

$$B_M = b_m, \tag{2.13}$$

$$A_j = \begin{cases} \frac{b_{j-1}-B_{j-1}}{b_j} & \text{if } j \neq 1 \text{ and } b_j \neq 0 \\ \frac{a_{i+1,j-1}-c_j}{B_j} & \text{if } j \neq 1 \text{ and } b_j = 0 \end{cases}, \tag{2.14}$$

for $i, j = 1, \ldots, s$ [4]. For low-storage explicit RK methods, A_1 will always equal to zero. While equations (2.12), (2.13) and (2.14) do give us a relationship between the Butcher and LSRK coefficients, they do not show that all RK methods have a low storage implementation. Therefore, we utilized those equations to build an algorithm that can transform an LSRK coefficient array into a Butcher Tableau.

2.2 LSRK to Butcher Tableau

In order to analyze any LSRK method, it is useful to convert the LSRK tableau to the equivalent Butcher Tableau. Converting Equation (2.10) to standard RK form in Equations (2.4) and (2.5) gives the conversion Algorithm 2.3.

```
function [a,b,c] = ConvertLSRK(A_vec,B_vec)
% A_vec = A vector of coefficients from low storage RK
% B_vec = B vector of coefficients from low storage RK

s = length(A_vec);
a = zeros(s,s);
b = zeros(1,s);
c = zeros(s,1);
b(1,s) = B_vec(s);

for p = s:-1:2
    b(1,p-1) = A_vec(p)*b(1,p)+B_vec(p-1);
end
for i = s:-1:1
    for j = s-1:-1:1
        if j>=i
            a(i,j) = 0;
        elseif i == j+1
            a(i,j) = B_vec(j);
        else
            a(i,j) = A_vec(j+1)*a(i,j+1)+B_vec(j);
        end
    end
end
for i = 1:s
    c(i) = sum(a(i,:),2);
end
end
```

Algorithm 2.3: Converts the LSRK method from Equation (2.10) *to standard RK form in Equations* (2.4) *and* (2.5) *and using* (2.6).

The two fourth order LSRK methods we use throughout this work are NRK14C [5] and RK54 [4]. Table 2.3 lists all of the coefficients for RK54 and Table 2.4 lists the coefficients for NRK14C.

0	0.149659021999229
-0.417890474499852	0.379210312999627
-1.19215169464268	0.822955029386982
-1.69778469247153	0.699450455949122
-1.51418344425716	0.153057247968152

Table 2.3: The A_j and B_j coefficients for RK54, from [4].

0	0.0367762454319673
-0.718801208672410	0.313629660755396
-0.778533117342157	0.153184869186903
-0.00532827966540440	0.00300970868181820
-0.855297993402928	0.332629379064611
-3.95641382457746	0.244025140535086
-1.57805753805874	0.371887923959228
-2.08370945525741	0.620412622158244
-0.748333418276161	0.152404317302874
-0.703286110656336	0.0760894927419266
0.00139170961176810	0.00776042140409780
-0.0932075369637460	0.00246472847553820
-0.951420047087595	0.0780348340049386
-7.11515716939226	5.50597772702696

Table 2.4: The A_j and B_j coefficients for NRK14C, from [5].

2.3 Order Conditions for RK Methods

By using the model ODE shown in Equation (2.1), the order conditions for RK methods come about starting with the Taylor series expansion of y in the neighborhood of t_n [1]. By comparing the Taylor series expansion of y to one step of the RK method, where the exact solution is used as the initial condition, the order conditions come from matching corresponding terms in the expansions. These order conditions must be satisfied for the method to be consistent and accurate. Here a more concise notation is introduced from Butcher [1]. For example, if we

assume (2.1) is autonomous with dimension m, then we differentiate it and we have

$$y''(t) = f'(y(t))y'(t). \tag{2.15}$$

To simplify this notation, we used the fact that $f = y'$, and we dropped the function arguments so that our derivative now corresponds to

$$y''(t) = f'f. \tag{2.16}$$

Remember that f' is a matrix since f is a vector valued function. When we take the derivative of (2.15), we get

$$y'''(t) = f''(f,f) + f'f'f, \tag{2.17}$$

where f'' is a bilinear operator with components

$$[f''(f,f)]_i = \sum_{j=1}^{m}\sum_{k=1}^{m} \frac{\partial^2 f_i}{\partial y_k \partial y_j} f_k f_j, \quad i = 1,\ldots,m. \tag{2.18}$$

The Taylor series expansion of the exact solution, $y(t+h)$, about t is

$$\begin{aligned} y(t+h) = y + hf + \frac{h^2}{2}f'f + \frac{h^3}{6}\Big[2f''(f,f) + f'f'f\Big] \\ + \frac{h^4}{24}\Big[6f'''(f,f,f) + 3f''(f'f,f) + f'f'f'f + 2f'f''(f,f)\Big] + O(h^5), \end{aligned} \tag{2.19}$$

where the higher order derivatives are multilinear operators defined similarly to $f''(f,f)$ in Equation (2.17) [1].

We define $z(h)$ to be one step of the RK method using the exact solution $y(t)$ as the

initial condition, where

$$z(h) = y + h\sum_{i=1}^{s} b_i F_i(h), \tag{2.20}$$

$$F_i(h) = f(Y_i(h)), \tag{2.21}$$

$$Y_i(h) = y + h\sum_{j=1}^{s} a_{ij} F_j(h). \tag{2.22}$$

In order to find the order conditions, we compare Equation (2.19) with the derivatives of $z(h)$. We drop the function arguments from Equations (2.20) through (2.22) for a more concise notation. The first five terms of the Taylor series expansion about zero of Equation (2.20) are

$$z(h) = z(0) + hz'(0) + \frac{h^2}{2}z''(0) + \frac{h^3}{6}z'''(0) + \frac{h^4}{24}z^{(4)}(0) + O(h^5). \tag{2.23}$$

For a method to be of global order p, the terms of the Taylor series of the exact solution $y(t+h)$ and the numerical solution $z(h)$ must match up to $O(h^{p+1})$. In Equations (2.19) and (2.23) we have only kept up through the fourth order terms because we are looking for order conditions for fourth order methods. If we compare equations (2.19) and (2.23), we begin to find the order conditions. We see that $z(0) = y$, which is the exact solution at t. Now we start to look at the higher order terms. For the first derivative of z, we have

$$z'(h) = \sum_{i=1}^{s} b_i F_i(h) + h\sum_{i=1}^{s} b_i F_i'(h). \tag{2.24}$$

At $h = 0$ the second term disappears, and using Equation (2.21), Equation (2.24) becomes

$$z'(0) = \sum_{i=1}^{s} b_i F_i(0) = \sum_{i=1}^{s} b_i f. \tag{2.25}$$

Again, comparing this result to the order h term in Equation (2.19), we find the first

order condition

$$\sum_{i=1}^{s} b_i = 1. \tag{2.26}$$

This condition is required for RK methods to be globally first order or consistent. To go any further, we must find the derivatives of Equations (2.21) and (2.22). Therefore we have

$$F_i'(h) = f' Y_i'(h), \tag{2.27}$$

$$Y_i'(h) = \sum_{j=1}^{s} a_{ij} F_j(h) + h \sum_{j=1}^{s} a_{ij} F_j'(h). \tag{2.28}$$

Continuing to take derivatives of Equation (2.20), we have

$$z''(h) = 2 \sum_{i=1}^{s} b_i F_i'(h) + h \sum_{i=1}^{s} b_i F_i''(h). \tag{2.29}$$

At $h = 0$ the second term disappears. Substituting Equation (2.28) into (2.27) and after using Equation (2.6) to convert a row sum of a_{ij} to c_i, Equation (2.29) becomes

$$z''(0) = \sum_{i=1}^{s} b_i F_i'(0) = \sum_{i=1}^{s} b_i c_i f' f. \tag{2.30}$$

Using $z''(0)$ and comparing the order h^2 terms in the two Taylor series expansions, we find the second order condition

$$\sum_{i=1}^{s} b_i c_i = \frac{1}{2}. \tag{2.31}$$

For the next derivative, we have

$$z'''(h) = 3 \sum_{i=1}^{s} b_i F_i''(h) + h \sum_{i=1}^{s} b_i F_i'''(h). \tag{2.32}$$

Now we need to find the second derivatives of Equations (2.21) and (2.22), which

are

$$F_i''(h) = f''(Y_i'(h), Y_i'(h)) + f' Y_i''(h), \tag{2.33}$$

$$Y_i'(h)' = 2\sum_{j=1}^{s} a_{ij} F_j'(h) + h\sum_{j=1}^{s} a_{ij} F_j''(h). \tag{2.34}$$

Substituting Equation (2.33) into (2.34), Equation (2.32) at $h = 0$ yields

$$z'''(0) = 3\sum_{i=1}^{s} b_i \left(c_i^2 f''(f, f) + 2\sum_{j=1}^{s} a_{ij} c_j f' f' f \right). \tag{2.35}$$

This results in two third order conditions

$$\sum_{i=1}^{s} b_i c_i^2 = \frac{1}{3} \tag{2.36}$$

and

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j = \frac{1}{6}. \tag{2.37}$$

As we have seen from the derivations of the first through third order conditions, this process is tedious. Deriving the fourth order conditions is more involved. We have to use the chain rule multiple times, which ends up generating four terms. Computing the fourth derivative of (2.20), we have

$$z^{(4)}(h) = 4\sum_{i=1}^{s} b_i F'''(h) + h\sum_{i=1}^{s} b_i F^{(4)}(h). \tag{2.38}$$

Our next step is finding the third derivative of Equations (2.21) and (2.22), which are

$$F_i'''(h) = f'''(Y_i(h), Y_i(h), Y_i(h)) + 3f''(Y_i''(h), Y_i''(h)) + f' Y_i'''(h), \tag{2.39}$$

$$Y_i'''(h) = 3\sum_{j=1}^{s} a_{ij} F_j''(h) + h\sum_{j=1}^{s} a_{ij} F_j'''(h). \tag{2.40}$$

Substituting Equations (2.39) and (2.40) into $z^{(4)}(h)$, yields

$$z^{(4)}(0) = 4\sum_{i=1}^{s} b_i\Big(c_i^3 f'''(f,f,f) + 2\sum_{j=1}^{s} c_i a_{ij} c_j f''(f'f,f) + 6\sum_{j=1}^{s}\sum_{k=1}^{s} a_{ij}a_{jk}c_k f'f'f'f + 3\sum_{j=1}^{s} a_{ij}c_j^2 f'f''(f,f)\Big). \tag{2.41}$$

Comparing this with the h^4 term from Equation (2.19), gives the fourth order conditions:

$$\sum_{i=1}^{s} b_i c_i^3 = \frac{1}{4}, \qquad \sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{8}, \tag{2.42}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{24}, \qquad \sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j^2 = \frac{1}{12}. \tag{2.43}$$

2.3.1 Using Trees to Derive Order Conditions

An equivalent and much less tedious method devised to derive order conditions for RK methods is by drawing rooted trees. We initially use rooted trees to derive the Taylor series coefficients in Equation (2.19). In this we label each node of the tree with $f^{(q)}$, where q equals the number of children. Later we use the rooted trees to compute the RK order conditions. Figure 2.1 shows a first order rooted tree that we label f, which corresponds to $y' = f$. To get the second order tree, we take a root and connect a copy of the first order tree to it. The leaf is then labeled as f while the root is labeled f' as in Figure 2.2. The second order tree then corresponds to $y'' = f'f$. There are two third order trees. We generate one by connecting a root to two copies of the first order tree and the other by connecting a root to the second order tree. Figure 2.3 shows these new trees. Together these two trees correspond to $y''' = f''(f,f) + f'f'f$. Using the previous trees, we form the order four trees in Figure 2.4.

The rooted trees correspond to each term of $y^{(4)} = f'''(f,f,f) + 3f''(f'f,f) + f'f'f'f + f'f''(f,f)$ in order from left to right. You may notice the three coefficient on the

Figure 2.1: Single node as the first rooted tree corresponding to Equation (2.26).

Figure 2.2: Second rooted tree corresponding to Equation (2.31).

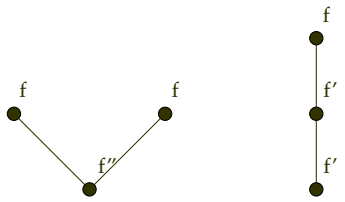


Figure 2.3: Rooted trees for the order three order conditions corresponding in order to Equation (2.36) *and* (2.37).

second term. This corresponds to number of ways we can label the tree with an ordered set and is shown in Figure 2.5. This is also known as the $\alpha(\tau)$ value for a given rooted tree τ.

The order conditions follow from the same trees; we just label them differently as in Figures 2.6 and 2.7. We label the root with the i index while each leaf is left unlabeled on the tree and takes the index of the node it is attached to. Any node(s) between the root and leaf are labeled in alphabetical order along the branches until all node are labeled. From each tree, we derive equations (2.26) through (2.37) and the fourth order equations in (2.42) and (2.43). As in Butcher [1], we represent the order conditions as

$$\Phi(\tau) = \frac{1}{\gamma(\tau)}, \tag{2.44}$$

where the index i represents order and τ is a single tree in the set of all trees. Here $\Phi(\tau)$ is defined from the tree τ to give the dependence of the order conditions on coefficients a, b and c. To do this we let the root of each tree start with b_i, and we

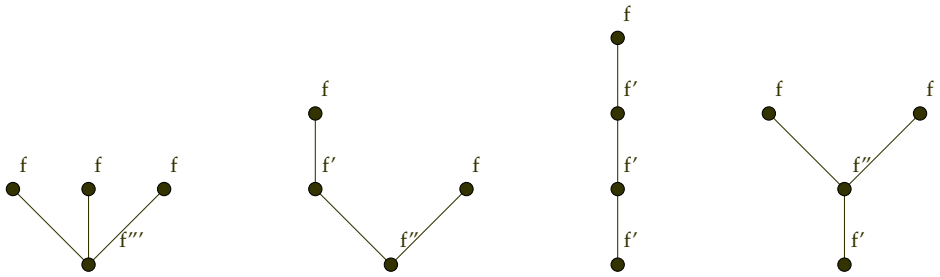


Figure 2.4: Rooted trees for the order four order conditions corresponding in order to Equations (2.42) *and* (2.43).

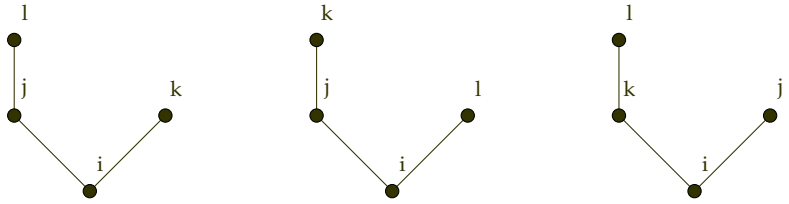


Figure 2.5: The number of ways we can label tree two from Figure 2.4.

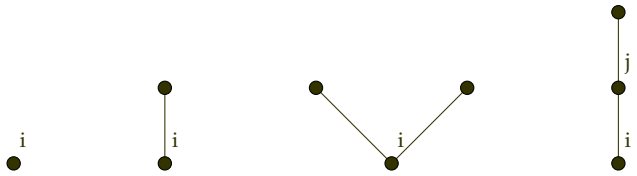


Figure 2.6: First, second and third order rooted trees with node labels for indexing.

let each leaf correspond to c, where c takes on the index of its parent. Any nodes between the root and terminal leaf are a coefficients, where a takes on the indices of itself and its parent. For example, the last two trees in Figure 2.7 use all of the rules to determine the order conditions. The second to last tree gives us b_i, a_{ij}, a_{jk} and c_k terms in as we move from root to leaf, while the last tree results in b_i, a_{ij} and two c_j terms as we see from Equation (2.43). Once we have all of the terms, we multiple them together and find their sum over the indices from one to s.

To find the number $\gamma(\tau)$, we assign a value of one to each node in the trees. We

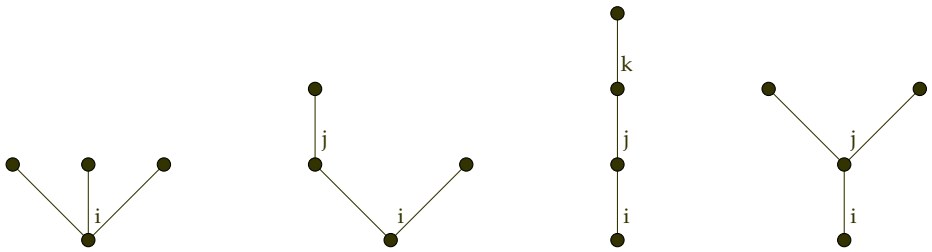


Figure 2.7: Order four rooted trees with node labels for indexing.

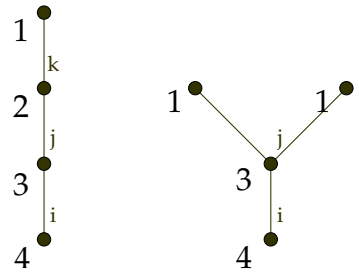


Figure 2.8: Example of rooted trees labeled to find the γ values.

then add the value on each node starting from the leaf nodes and traveling down to the root. The total sum on the root equals the order of the tree. An example is provided in Figure 2.8. Multiplying all of the numbers from the tree gives the number $\gamma(\tau)$, which for the trees in Figure 2.8 equals 24 and 12. These match up with Equation (2.43). The other order conditions follow accordingly.

2.3.2 Fifth Order Conditions

As seen previously, algebraically deriving the order conditions for RK methods is no trivial task. Therefore, we will rely on the tree derivation of the fifth order condition. To generate the fifth order trees we connect a root to copies of lower order trees. We show all of the fifth order trees in Figures 2.9–2.11. These figures are labeled to determine $\gamma(\tau)$. We derive $\Phi(\tau)$ as shown in Section 2.3.1. The order conditions for the trees in Figure 2.9 are

$$\sum_{i=1}^{s} b_i c_i^4 = \frac{1}{5}, \quad \sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i^2 a_{ij} c_j = \frac{1}{10}, \quad \sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i a_{ij} c_j^2 = \frac{1}{15}. \tag{2.45}$$

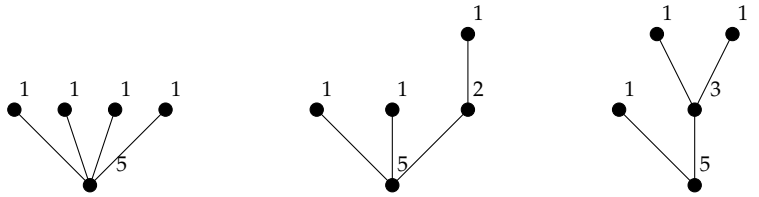


Figure 2.9: Order five order conditions one through three labeled to find γ.

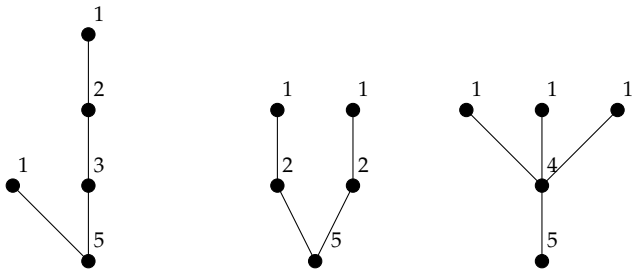


Figure 2.10: Order five order conditions four through six labeled to find γ.

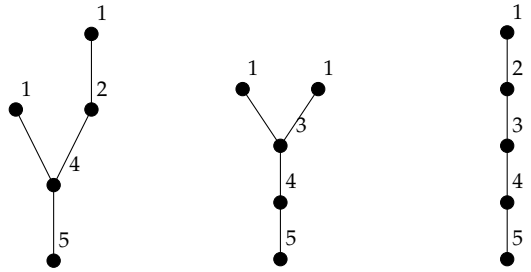


Figure 2.11: Order five order conditions seven through nine labeled to find γ.

The order conditions for the trees in Figure 2.10 are

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i c_i a_{ij} a_{jk} c_k = \frac{1}{30}, \quad \sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} a_{ik} c_j c_k = \frac{1}{20}, \quad \sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j^3 = \frac{1}{20}. \quad (2.46)$$

The order conditions for the trees in Figure 2.11 are

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} c_j a_{jk} c_k = \frac{1}{40}, \quad \sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} a_{jk} c_k^2 = \frac{1}{60}, \\ \sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{s} b_i a_{ij} a_{jk} a_{kl} c_l = \frac{1}{120}. \tag{2.47}$$

2.4 Checking Order Conditions for LSRK Methods

Now that we have the order conditions, we verify that our chosen LSRK schemes indeed satisfy. Since both RK54 and NRK14C are fourth order methods, we check through the fourth order conditions laid out in Section 2.3. We use Algorithm 2.4 to check all the order conditions.

```
function [c,norms] = OrderCondition(s,p,a,b,tol)
% s = number of stages
% p = order of method
% a = tableau
% b = weights of a
% tol = tolerance

c = sum(a,2);

condition_vector = zeros(1,17);
for i = 1:s % single summation order conditions
    condition_vector(1) = condition_vector(1) + b(i);
    condition_vector(2) = condition_vector(2) + b(i)*c(i);
    condition_vector(3) = condition_vector(3) + b(i)*(c(i)^2);
    condition_vector(5) = condition_vector(5) + b(i)*(c(i)^3);
    condition_vector(9) = condition_vector(9) + b(i)*(c(i)^4);
    for j= 1:s % double summation order conditions
        condition_vector(4) = condition_vector(4) + b(i)*a(i,j)*c(j);
        condition_vector(6) = condition_vector(6) + b(i)*c(i)*a(i,j)*c(j);
        condition_vector(7) = condition_vector(7) + b(i)*a(i,j)*(c(j)^2);
        condition_vector(10) = condition_vector(10) + b(i)*(c(i)^2)*a(i,j)*c(j);
        condition_vector(11) = condition_vector(11) + b(i)*c(i)*a(i,j)*(c(j)^2);
        condition_vector(14) = condition_vector(14) + b(i)*a(i,j)*(c(j)^2);
        for k = 1:s % triple summation order conditions
            condition_vector(8) = condition_vector(8) + b(i)*a(i,j)*a(j,k)*c(k);
            condition_vector(12) = condition_vector(12) + b(i)*c(i)*a(i,j)*a(j,k)*c(k);
            condition_vector(13) = condition_vector(13) + b(i)*a(i,j)*a(i,k)*c(j)*c(k);
            condition_vector(15) = condition_vector(15) + b(i)*a(i,j)*c(j)*a(j,k)*c(k);
            condition_vector(16) = condition_vector(16) + b(i)*a(i,j)*a(j,k)*(c(k)^2);
```

```
            for l = 1:s % quadruple summation order conditions
                condition_vector(17) = condition_vector(17) ...
                                     + b(i)*a(i,j)*a(j,k)*a(k,l)*c(l);
            end
        end
    end
end
conditionsRHS = [1 1/2 1/3 1/6 1/4 1/8 1/12 1/24 1/5 ...
                 1/10 1/15 1/30 1/20 1/20 1/40 1/60 1/120];
for z = 1:length(condition_vector)
    if abs(condition_vector(z)-conditionsRHS(z)) <= tol
        success = ['Passes order condition ',num2str(z),'.'];
        display(success)
    end
end
% Truncation Error Coefficient
Tc = 17;
phi = condition_vector(1,1:Tc);
gamma = conditionsRHS(1,1:Tc);
alpha = [1 1 1 1 1 3 1 1 1 6 4 4 3 1 3 1 1];
rho = [1 2 3 3 4 4 4 4 5 5 5 5 5 5 5 5 5];
upsilon = (phi.*gamma-1).*(alpha./factorial(rho));
norms = [norm(upsilon,2);norm(upsilon,Inf)];
```

Algorithm 2.4: Checks order conditions for 1^{st} through 5^{th} order methods.

If we satisfy all of the order conditions up to order four for a certain user defined tolerance, then we have a valid LSRK method. Now we can test RK54 and NRK14C against RK4.

2.5 Truncation Error Coefficients

One way to compare RK methods used in Chapter 3, is with truncation error coefficients (TEC). The TEC for the tree τ is defined as

$$\Upsilon(\tau) = (\Phi(\tau)\gamma(\tau) - 1)\frac{\alpha(\tau)}{\rho(\tau)!}. \tag{2.48}$$

We generate this equation by matching terms and equations from the Taylor series expansions of the exact and numerical solutions [6]. We saw all of the variables used in Equation (2.48) in Section 2.3 except $\rho(\tau)$, which is the order of the tree.

Equation (2.48) gives us value corresponding to each tree. We form a vector where

each component is Equation (2.48) evaluated for one tree; we consider the l_2 norm and the l_∞ norm of this vector. Recall that $\Phi(\tau)$ depends on the coefficients of the RK method. The closer that $\Phi(\tau)\gamma(\tau)$ is to one the smaller the magnitude of the TEC. Furthermore, $\Gamma(\tau)$ will be zero when an order condition is satisfied, so any fourth order scheme will have zero TEC values for the first eight trees. This does not tell us the whole story though as we will see in the next chapter.

Algorithm 2.4 contains the TEC calculation at the end of the code. To test the implementation, we compared the TEC values we found to those in Cameron's work [6] for the Bogacki-Shampine 3(2) method [7]. Since our code reproduced published results, we have confidence in our implementation.

2.6 Stability Region

Consider the one-dimensional model ODE

$$y' = \lambda y, \tag{2.49}$$

where the exact solution is

$$y = Ce^{\lambda t}. \tag{2.50}$$

As described in Ascher and Petzold [8], we find the values of $z = h\lambda$ where the numerical solution to Equation (2.49) does not increase in magnitude. Starting with Equations (2.4) and (2.5) and using the model ODE, a full single step update to go from y_n to y_{n+1} we now have

$$y_{n+1} = \left[1 + z + \frac{z^2}{2} + \ldots + \frac{z^p}{p!} + \sum_{j=p+1}^{s} z^j b a^{j-1} \mathbb{1}\right] y_n = \left[\sum_{j=0}^{s} \xi_j z^j\right] y_n, \tag{2.51}$$

$$\xi = \left[1, \frac{1}{2!}, \frac{1}{3!}, \ldots, \frac{1}{p!}, ba^{j-1}\mathbb{1}, \ldots, ba^{s-1}\mathbb{1}\right]. \tag{2.52}$$

If we let K be coefficient in front of y_n from Equation (2.51), then $|K|$ is the growth factor of our solution. We can then say that if $|K|$ is greater than one, then the modulus of the solution grows exponentially. If $|K|$ equals one, then the modulus of the solution does not change. However, if $|K|$ is less than one, the modulus of

the solution decays, which is called absolutely stable [8]. With this in mind, we can plot the region where we would expect the solution to remain stable. Figure 2.12 shows the region where each method will remain stable if their respective z values are within the boundary. If z are outside of the boundary, the solution is not stable.

Equation (2.51) along with MATLAB's `contour` plot function led us to build the following algorithm to plot stability regions for each LSRK method used within this work.

```
% Create the R(z) equation to determine the stability region
load('NRK14C')
A_vec = NRK14C(:,1);
B_vec = NRK14C(:,2);
[a,b,c] = ConvertLSRK(A_vec,B_vec);
p = 4;
s = length(A_vec);

% Define the mesh
xv = linspace(-20,10,100);
yv = linspace(-11,11,100);
[x,y] = meshgrid(xv,yv);

z = x + 1i*y;
w = 0;

Rz = 1 + z + z.^2/2 + z.^3/6 + z.^4/24;
for  k = (p+1):s
  w = w + z.^k*(b*a^(k-1)*ones(s,1));
end

Rz = Rz + w;
contour(x,y,abs(Rz),[1 1],'b')
```

Algorithm 2.5: Algorithm that plots the stability region for a given LSRK scheme.

Figure 2.12 shows the scaled stability regions for each method. We scaled each stability region by the number of stages for that method. We scale the plot by the number of stages as this is the number of times f must be evaluated in the schemes. You will notice from Figure 2.12 that RK4 includes a larger portion of imaginary axis, but NRK14C includes a much larger stability region. Knowing the region of absolute stability for each method used to solve an ODE is useful for determining an appropriate time step size that will yield a useful solution. This proves to be a

useful property for the problems solved in the following sections.

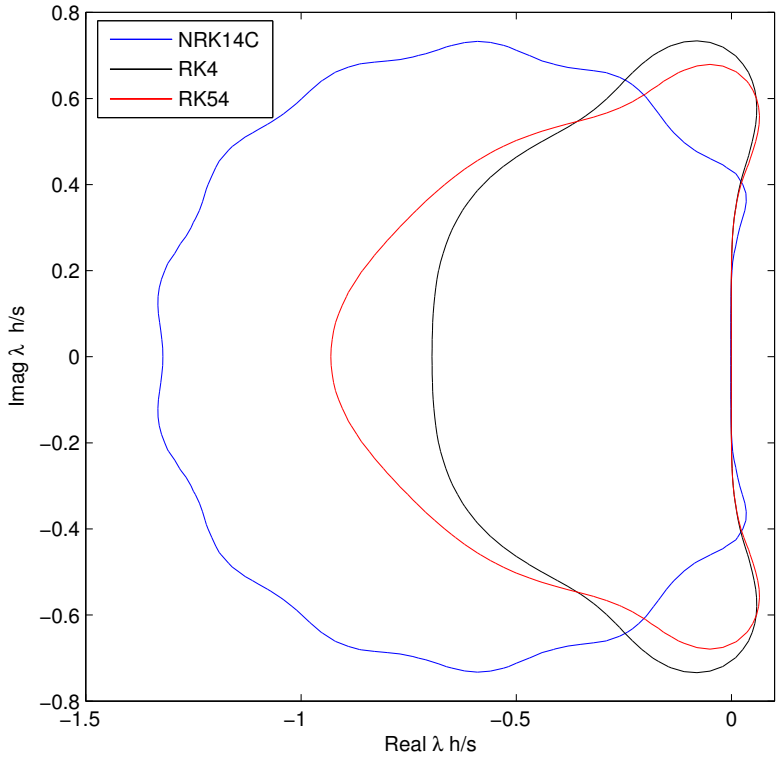


Figure 2.12: This figure shows the scaled stability regions for NRK14C, RK54 and RK4. Any eigenvalues of the problem solved that are within the region will remain stable.

CHAPTER 3:
Testing RK Methods

This chapter shows two examples of using the RK methods from Chapter 2 to solve a system of ODEs. First, we analyze the results for a simple ODE, which we used for all calculations in the first section. Secondly, we analyze the results of the LSRK methods used to solve Maxwell's equations in 2D. This chapter shows how well each RK method solves the equations by comparing the numerical results to the exact solution. We also show how many operations it takes each method to achieve a specific level of accuracy. The last metric we use to compare the methods is how large the time step can be in each case.

3.1 Comparing LSRK to RK4 Using an ODE

To understand the power of using a low storage method, we start with the following ODE and initial condition

$$y' = Fy = \begin{pmatrix} 0 & 20 \\ -20 & 0 \end{pmatrix} y, \quad y(0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

We compute the numerical solution of this ODE from the initial condition, $t = 0$, to a final time, $t = 10$. We use RK4, Algorithm 2.1, and LSRK, Algorithm 2.2, to solve the ODE. To compare the methods, we use the l_2 norm of the difference between the exact solution,

$$y(t) = \begin{pmatrix} \sin(20t) \\ \cos(20t) \end{pmatrix},$$

and the numerical solution at $t = 10$. We plot the error against the time step size to see if the results are as expected. Figure 3.1 shows the log-log error plot for NRK14C, RK54 and RK4. We use a log-log scale to determine the order. From Figure 3.1 we have fourth order convergence for all three methods, and we see that each simulation has order one error around $h = 0.1$. Upon further investigation,

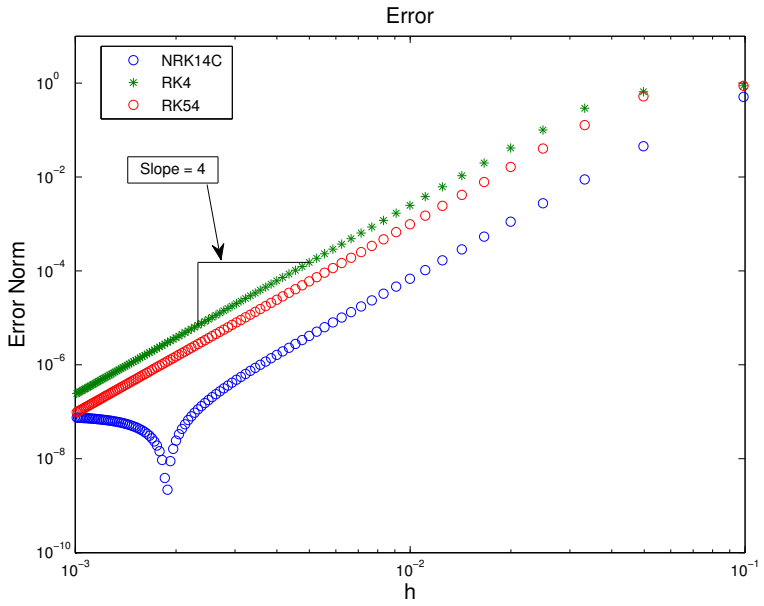


Figure 3.1: Error between the exact and numerical solution for various time step sizes. Also shows that our methods have fourth order convergence.

we see in Figure 3.2 that the error for each method grows exponentially at some critical h. However, the range of stable time step sizes is different for each method. For NRK14C method, we note that we can take a larger time step while remaining stable. Similarly, RK54 allows a larger stable time step than RK4. As mentioned at the end of Chapter 2, the stability region should give us an indicator of the time step size required to have a stable solution. If we look at the unscaled stability regions in Figure 3.3, it is immediately apparent that NRK14C has the largest stability region followed by RK54 and RK4 has the smallest. Figure 3.2 is then expected given the time step sizes for each method and their respective stability regions.

The computational cost of an RK method is related to the number of times f must be called. This is especially true for high dimensional ODEs such as those from the discretization of PDEs. To examine this cost, we show the operation count (number of f evaluations) versus the step size in Figure 3.4. If we use a larger time step, we decrease the number of times we evaluate f, which reduces the computational cost of solving the problem. Let us now look at what the potential savings are for this problem. Table 3.1 shows the number of operations required to reach a certain

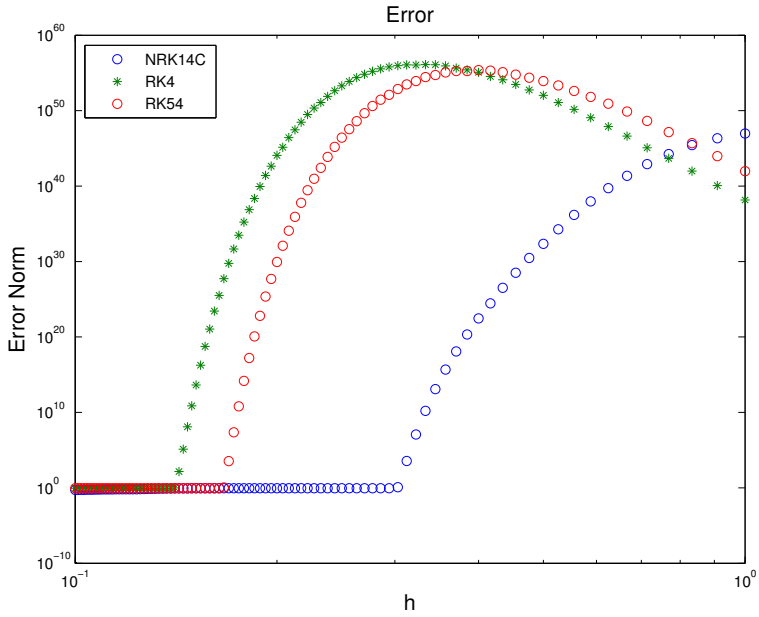


Figure 3.2: Log-log plot of the error for $h > 0.1$ where the stability of the time step is different for each method.

	RK4		**RK54**		**NRK14C**	
Error	h	Operations	h	Operations	h	Operations
0.01	0.01408	2841	0.01786	2800	0.03448	4060
0.0001	0.004525	8840	0.005682	8800	0.01099	12739
0.000001	0.001437	27836	0.001805	27701	0.003571	39205

Table 3.1: Shows the number of operations each method takes to reach the error level in column one.

level of error for each of the three methods used to solve the ODE. You can see that NRK14C requires about 30% more operations than RK4 to reach the same error level. Even though RK54 has one more stage evaluation than RK4, it takes fewer operations to attain the same error level. This savings is one reason why some low storage methods are an attractive alternative to the standard RK4.

3.2 Solving Maxwell's Equation in 2D

As in the previous section, we want to analyze the differences between RK methods. In this section we will focus on a more complex problem, namely solving Maxwell's

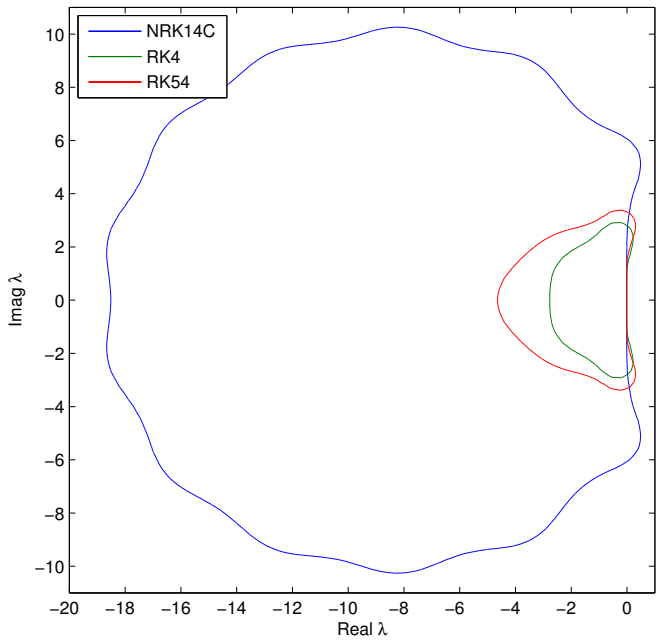


Figure 3.3: Unscaled stability region for RK4, RK54 and NRK14C.

Equation in 2D

$$\begin{aligned} \mu \frac{\partial \tilde{H}_x}{\partial \tilde{t}} &= -\frac{\partial \tilde{E}_z}{\partial \tilde{y}}, \\ \mu \frac{\partial \tilde{H}_y}{\partial \tilde{t}} &= -\frac{\partial \tilde{E}_z}{\partial \tilde{x}}, \\ \varepsilon \frac{\partial \tilde{E}_z}{\partial \tilde{t}} &= \frac{\partial \tilde{H}_y}{\partial \tilde{x}} - \frac{\partial \tilde{H}_x}{\partial \tilde{y}}. \end{aligned} \tag{3.1}$$

Here, we have the magnetic fields $\tilde{H}_x$ and $\tilde{H}_y$, the electric field $\tilde{E}_z$, magnetic permeability μ and the electric permittivity ε, which are all functions of $\tilde{x}$, $\tilde{y}$, $\tilde{z}$ [9]. In order to discretize Maxwell's Equation in 2D, we utilized the Discontinuous Galerkin (DG) codes available from Hesthaven and Warburton [9]. This method uses a method-of-lines (MOL) approach where we use high order polynomials on triangular elements for the spatial discretization. For the temporal discretization, we compare RK54 and NRK14C. We used the domain $[-1,1]\times[-1,1]$ meshed as

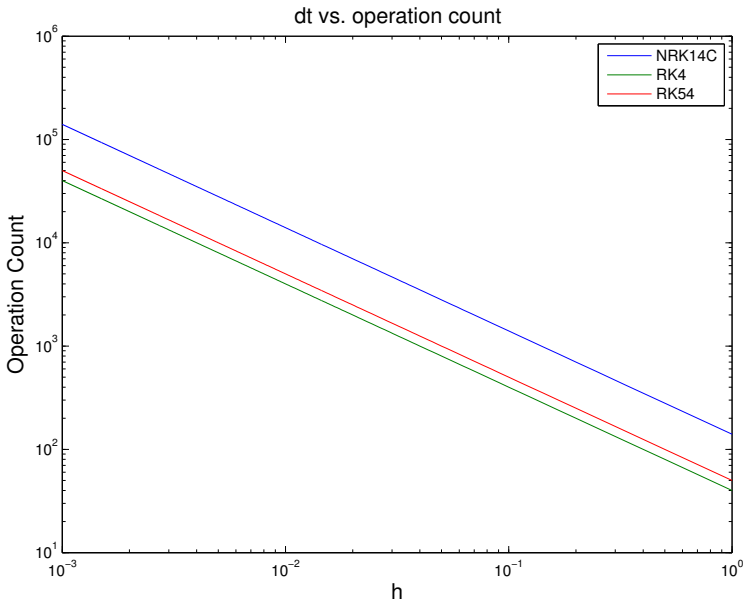


Figure 3.4: Inverse relationship between the time step and number of operations.

shown in Figure 3.7(a) and integrated from $t = 0$ to $t = 1$. We define the error using the L_2 norm of the difference between the solver's solution vector for E_z and the exact solution $E_z = \sin(\pi x)\sin(\pi y)\cos(\omega t)$, where $\omega = \pi\sqrt{2}$ and $t = 1$.

3.2.1 Results

By changing only the size of the time step for each error calculation, we identified where temporal or spatial error dominated. We also ran the code with different polynomial orders ($N = 4,6,8,10$) to show how the spatial error interacts with time step size. The solution converged for a small time step size, but it becomes unstable when the time step size increased. The time step size where the solution becomes unstable is different for each polynomial order. From Table 3.2 we discern what level of accuracy the methods achieve for a stable time step size. A graphical representation of this data is also shown in Figures 3.5 and 3.6. Even though we are able to take a larger step size with NRK14C, RK54 is more efficient because it has a lower operations count and target error for each polynomial order as seen in Table 3.2.

		RK54		NRK14C	
Polynomial Order	Error	h	Operations Count	h	Operations Count
$N = 10$	1.0×10^{-10}	0.0020606	2430	-	-
	1.0×10^{-9}*	-	-	0.0071174	1974
	1.0×10^{-9}	0.0036701	1365	0.0093645	1498
	1.0×10^{-8}	0.0053442	940	0.0132506	1064
$N = 8$	1.0×10^{-10}	0.0020439	2450	-	-
	1.0×10^{-9}*	-	-	0.0071174	1974
	1.0×10^{-9}	0.0036659	1365	0.0093645	1498
	1.0×10^{-8}	0.0065232	770	0.0132506	1064
	1.0×10^{-7}	-	-	0.0226398	630
$N = 6$	1.0×10^{-7}	0.0115735	435	0.0225682	630
	1.0×10^{-6}	-	-	0.0403282	392
$N = 4$	1.0×10^{-6}	-	-	0.0628060	224
	1.0×10^{-5}	0.0219035	230	-	-

*Table 3.2: Number of right hand side calls for the specified time step size and polynomial order. *This error level happens before the NRK14C plot anomaly.*

3.2.2 Ensuring Spatial Error Dominance

The previous section used the mesh as shown in Figure 3.7(a). To ensure spatial error dominance, which is typically the case when solving Maxwell's Equations using the MOL, we decreased the mesh resolution to what is shown in Figure 3.7(b). Changing the mesh decreases the spatial resolution for this problem. Let us solve Maxwell's Equations using the coarse mesh, but now we only use polynomials of order $N = 4$ and $N = 6$. For $N = 4$ spatial error dominates the solution error. The solution remains stable for each method up to a certain time step size as we have discussed previously. For this problem, the limit of stability for NRK14C was a time step of $h = 0.1708$ for $N = 4$ and $h = 0.09849$ for $N = 6$. For RK54 the limit of stability was a time step of $h = 0.04125$ for $N = 4$ and $h = 0.02398$ for $N = 6$. If we calculate the operations required for each method, we find that NRK14C is more efficient for both polynomial orders as shown in Table 3.3. Thus, even though NRK14C has larger error when the solution is well resolved spatially, it may be the preferred method for simulations which are under-resolved, which is arguably where most practical calculations are preformed.

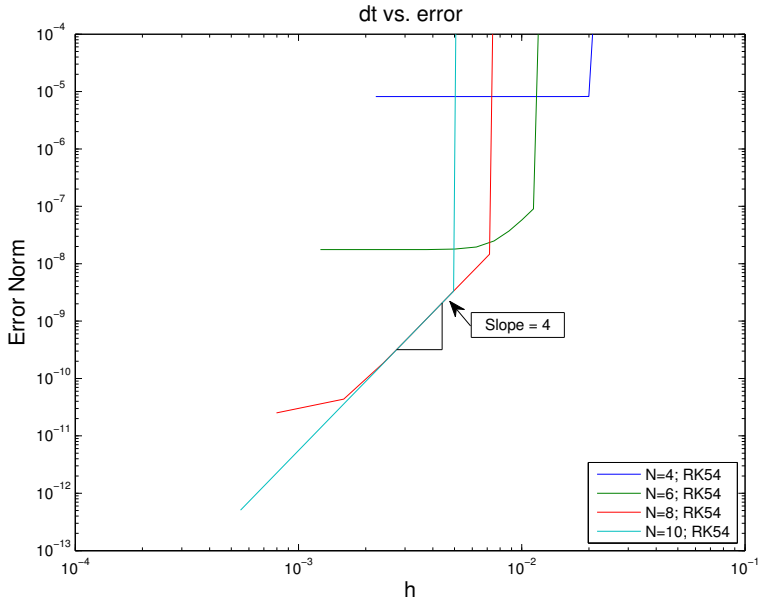


Figure 3.5: RK54 with $N = 4, 6, 8, 10$ polynomial orders.

	RK54		NRK14C	
Polynomial Order	h	Operations Count	h	Operations Count
$N = 4$	0.04125	120	0.1708	84
$N = 6$	0.02398	210	0.09849	140

Table 3.3: Operations count for RK54 and NRK14C using the coarse mesh and polynomials order N= 4, 6.

3.2.3 Stability Region and Eigenvalues

Now that we have the time step values for which the error begins to grow exponentially, we take another look at the stability region. The spatial discretization operator used above yields the semi-discrete ODE system $y' = Ly$. The eigenvalues of L that control the stability of the problem. We found the eigenvalues for $N = 4$ with the coarse mesh using Algorithm C.1. Figures 3.9 and 3.10 show how the scaled eigenvalues, $h\lambda$, fill the stability region. If we increased the time step size in either plot by the tiniest fraction, the scaled eigenvalues move out of the stability region and the error grows. If we instead chose a much smaller time step size, the scaled eigenvalues remain in the stability region and we still have a stable solution.

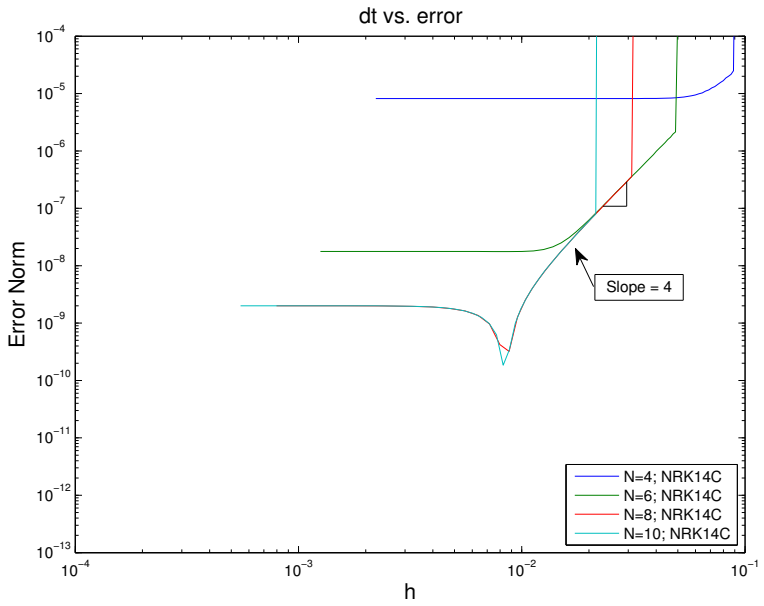


Figure 3.6: NRK14C with $N = 4, 6, 8, 10$ polynomial orders.

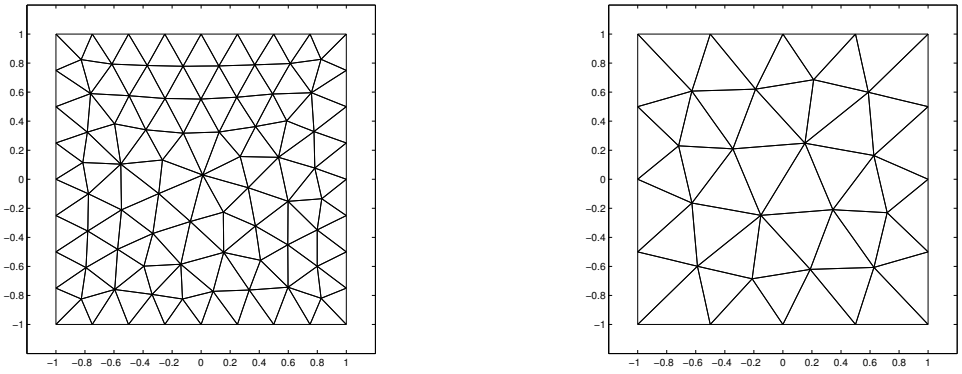
Method	l_2 **Norm**	l_∞ **Norm**
RK4	0.01298	0.00833
RK54	0.00787	0.00556
NRK14C	0.00560	0.00556

Table 3.4: Table of TECs for RK4, RK54 and NRK14C.

3.3 Truncation Error Coefficients and Conclusions

For each of the methods explored in this chapter, we compute the associated TECs. As you can see from Table 3.4, the l_2 norm of the TEC vector reveals that NRK14C has the lowest value while the l_∞ norm shows that RK54 and NRK14C are the same. With the l_2 norm, RK4 has an order of magnitude larger value. From only the TEC information, we might conclude that NRK14C is the best method. However, the TECs are the coefficients of the error terms from the Taylor series. Since these are fourth order methods, the leading error terms are order h^5. The TEC is smaller for NRK14C, but when multiplied by h^5, it becomes clear that large time steps lead to larger errors.

For the examples in this chapter, the analysis shows that LSRK is more efficient



(a) Example mesh used for initial simulations.

(b) An example of the coarse mesh used for a less resolved solution.

Figure 3.7: Comparison of the two meshes used in this problem. Notice that the coarser mesh on the right forced the problem to be dominated by spatial error, which is typical for the MOL approaches to solving Maxwell's Equations, from [9].

than RK4. We take advantage of storing fewer registers over the standard RK implementation to increase efficiency. We investigated the potential benefits LSRK methods by comparing RK54 and NRK14C when solving Maxwell's Equation. We saw that in some cases NRK14C is more efficient than RK54 particularly when the problem is under-resolved. Additionally, we pay for using a larger time step when we consider the higher order error terms. All of this analysis shows that we must consider both error and stability together when choosing a method for a particular problem.

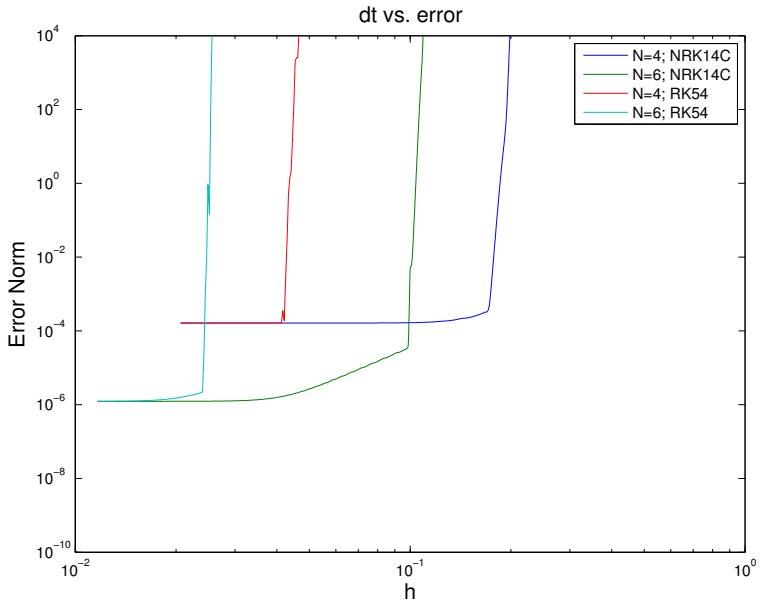


Figure 3.8: Shows error versus h with $N = 4$ and 6 polynomial order using the mesh from Figure 3.7(b).

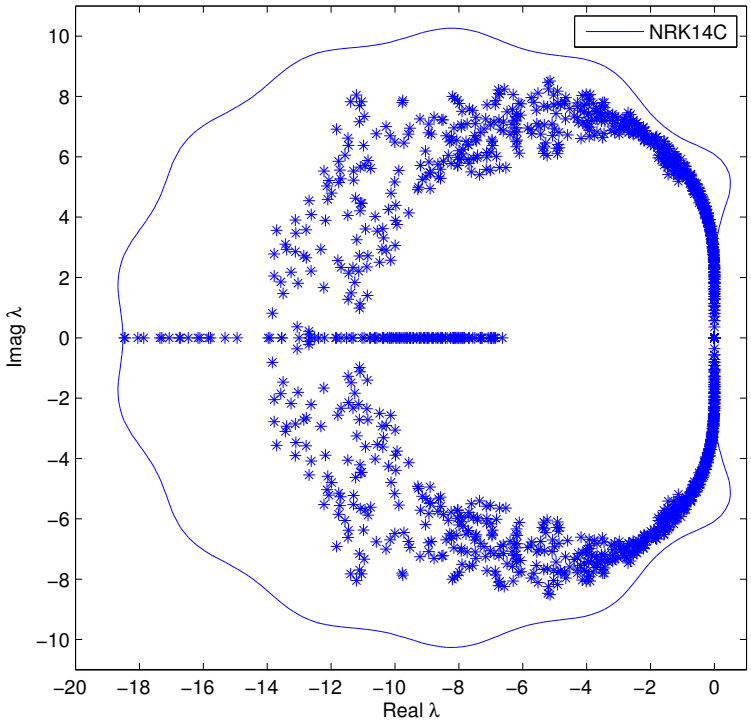


Figure 3.9: The unscaled NRK14C stability region with scaled eigenvalues ($N = 4$ and $h = 0.1708$).

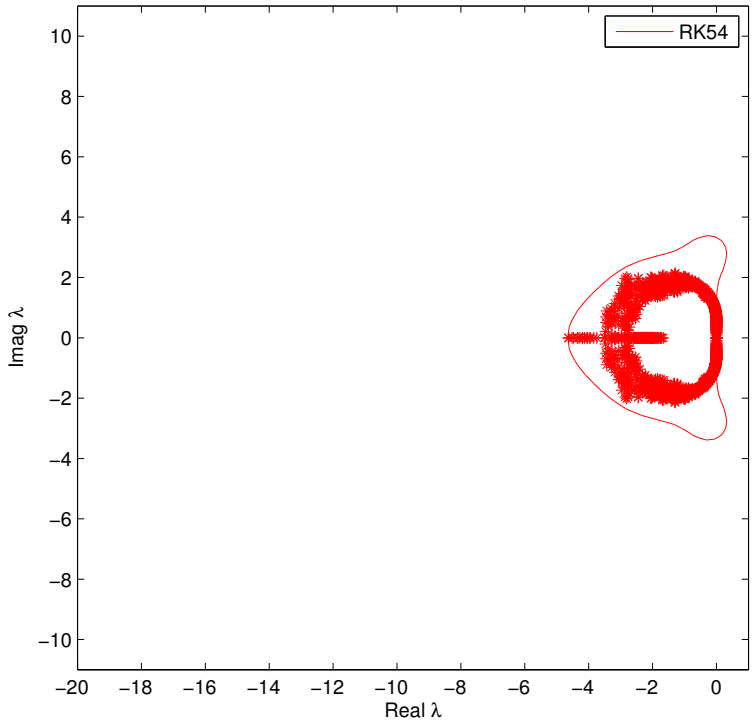


Figure 3.10: The unscaled RK54 stability region with scaled eigenvalues ($N = 4$ and $h = 0.04125$).

CHAPTER 4:
Optimization for a New LSRK Method

Having analyzed the potential benefits of LSRK methods, we present a method for discovering new LSRK methods. In this chapter, we show how the MATLAB optimization solver `fmincon` is used to find new LSRK coefficients. The optimization problem we consider is

$$\begin{aligned} \arg\min_{x} \quad & \mathcal{F}(x) \\ \text{subject to} \quad & \mathcal{E}(x) = 0, \\ & l < x < u, \end{aligned} \tag{4.1}$$

where

$$x = \begin{pmatrix} A_2 \\ A_3 \\ \vdots \\ A_s \\ B_1 \\ B_2 \\ \vdots \\ B_s \end{pmatrix}. \tag{4.2}$$

Our strategy for finding new methods is to minimize $\mathcal{F}(x)$, a measure of the fifth order conditions, while enforcing the first though fourth order conditions as equality constraints, $\mathcal{E}$. The thought process behind this was that if we can reduce the error of the fifth order terms, the new LSRK method would be more accurate. We discuss the bounds l and u below. The collection of MATLAB codes using `fmincon` are in Appendix D. We also demonstrate how the stability region plays a role in shaping the performance of the method. We add a constraint to ensure we achieve a similarly large stability region akin to the one for the NRK14C used previously in this work.

4.1 Implementing the Optimization without Shape Constraints

To begin the optimization, `fmincon` requires a number of constraints, options and starting guesses. We changed the default options for maximum function evaluations and maximum iterations to 600000 and 30000, respectively. We also changed the tolerance on the constraints and function to 1×10^{-10}. We start by defining the initial guess x, the LSRK coefficients, for the solver. We started with the coefficients from NRK14C and ran separate optimizations scaling one coefficient at a time by 0.9. The bounds, l and u, we used are ±20, which are a little larger than the coefficients from NRK14C.

The stability regions for a few of the 14 stage methods that `fmincon` found are compared with NRK14C in Figure 4.1. Figure 4.1 shows three of the methods discovered, but each one has a completely different shape. Compared to NRK14C, each has a smaller region of stability. Therefore, the three new methods would require a smaller time step than that of NRK14C, which would be less efficient.

4.2 New LSRK

Now that we know we are able to use `fmincon` to find new 14 stage LSRK methods, we want to match the large stability region of NRK14C. In the paper by Niegemann et al. [5], the authors show a set of values for ξ_i from Equation (2.51) that force a method to take on a specific shape. These are included along with the order conditions as equality constraints in the optimization problem. For the 14 stage method, we use their ξ_i values, which we include in Algorithm D.3. This ensures that any new 14 stage LSRK method we find has the now familiar circular stability region for NRK14C. Including all of the new constraints for shape, we run Algorithm D.1. We found that perturbing individual coefficients from NRK14C for the initial guess x_0 gave us good results. After many initial guesses, one method stood out. The new method in Table 4.2 satisfied the first through fourth order conditions to the same tolerance as NRK14C. We call the method ORK14, which is short for optimized RK 14 stage. If we plot the stability region for both NRK14C and ORK14, we see that they are virtually indistinguishable. If we magnify the boundary region, there are

i	ξ_i
1	1
2	1/2
3	1/3
4	1/6
5	1/4
6	1/8
7	1/12
8	1/24
9	$8.0971474827892589\times10^{-3}$
10	$1.2380169165300218\times10^{-3}$
11	$1.4920544370587013\times10^{-4}$
12	$1.4105197862197588\times10^{-5}$
13	$1.0338060754675449\times10^{-6}$
14	$5.7551620074656494\times10^{-8}$
15	$2.3518316167532871\times10^{-9}$
16	$6.6527970264862166\times10^{-11}$
17	$1.1639946786449694\times10^{-12}$
18	$9.4910013085549050\times10^{-15}$

Table 4.1: The shape constraints from Equation (2.51).

portions of the plot where one stability region is fraction larger than the other, but this is a negligible. Now we must look to see if there is any advantage to using this new LSRK method by using it to solve the Maxwell's Equation from Chapter 3. We run the code again using both NRK14C and ORK14 to graph the error versus the time step size, which results in Figure 4.3. There is no discernible difference between the methods as shown in Figures 4.2 and 4.3.

4.3 TECs and Conclusions

Now we calculate the TEC norm for our new method for another method of comparison. The TEC information for ORK14 along with previously discussed methods is in Table 4.3. We postulate that NRK14C and ORK14 are essentially the same despite having different coefficients. What we take away from this analysis is that it is difficult to find a 14 stage method and even harder to find one that matches the performance of NRK14C. The fact that we did find another method

0	0.011836003073137
-0.690728081645704	0.351015813936534
-1.234996309208810	0.063118128360741
-0.004609176627937	0.003932652530549
-0.858269465072828	0.455569623390910
-3.958588101464950	0.233901712708693
-1.650438770236550	0.487748443848318
-2.259100953376070	0.602337879585889
-0.536934229063860	0.095602003443996
-0.654598864540301	0.122569752673352
0.002105760404814	0.010112843394072
-0.110069481428190	0.003775965860034
-0.970342181377515	0.119011380003210
-7.112965306572070	5.520471813276850

Table 4.2: The A_j and B_j coefficients for ORK14.

Method	l_2 **Norm**	l_∞ **Norm**
RK4	0.01298	0.00833
RK54	0.00787	0.00556
NRK14C	0.00560	0.00556
ORK14	0.00560	0.00556

.

Table 4.3: Table including new ORK14 TEC information.

indicates that there are other 14 stages methods out there for discovery. In this chapter we relied only on MATLAB's `fmincon` with the options changed to using a higher tolerance and allowing more function evaluations and iterations. Other approaches may yield better results.

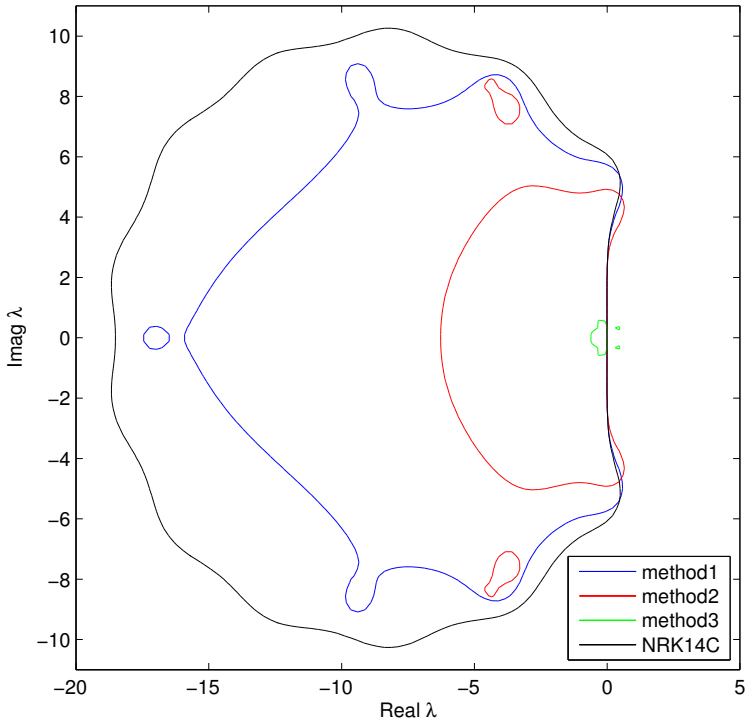


Figure 4.1: Three different LSRK methods found using `fmincon`.

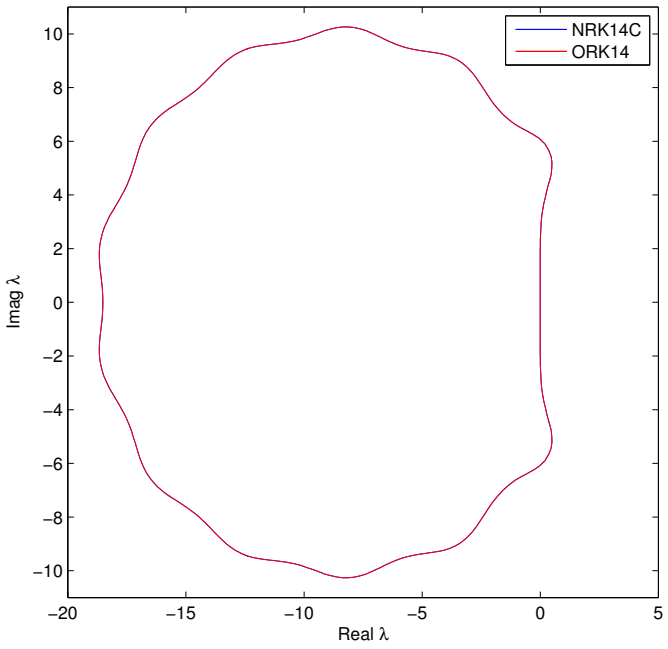


Figure 4.2: Stability region plot for NRK14C and ORK14. Note that the lines are on top of each other.

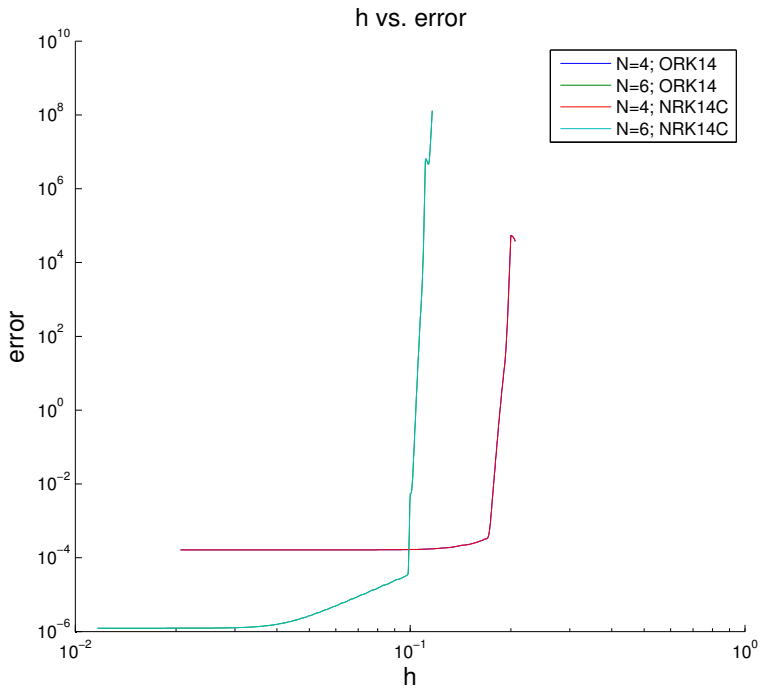


Figure 4.3: Error versus time step size for NRK14C and ORK14. Note that the lines are on top of each other.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Half-Explicit Methods

Now that we have developed a method for finding and testing LSRK methods, we want to explore another class of RK methods called half-explicit Runge-Kutta (HERK) methods. The HERK methods are of use when solving differential-algebraic equations (DAEs). We introduce both DAEs as well as HERK methods and show how HERK methods solve DAEs. We give a new low storage half-explicit method and solve a DAE using it. The associated algorithms and MATLAB code for this chapter are located in Appendicies E and F.

5.1 Differential-Algebraic Equations

A DAE is essentially an ODE with algebraic constraints. The addition of a constraint is what makes this a DAE [10]. For example, we model some mechanical systems using DAEs. We focus on index-2 DAEs, which are systems of equations with the general form

$$y' = f(y,z), \tag{5.1}$$
$$0 = g(y), \tag{5.2}$$

where f and g are smooth enough and $g_y(y)f_z(y,z)$ is nonsingular in the neighborhood of the solution. The index of the problem refers to the number of times we differentiate the constraint to reduce the problem to an ODE [11]. For example, if we take the first derivative of Equation (5.2), we get

$$g'(y)y' = 0 \rightarrow g'(y)f(y,z) = 0. \tag{5.3}$$

We simplify this expression by dropping the function arguments. Next, we take the second derivative of Equation (5.2) and we get

$$g''f^2 + f_y y' + f_z z' = 0 \rightarrow g''f^2 + f_y f + f_z z' = 0. \tag{5.4}$$

We write the DAE constraint as

$$z' = -\frac{g'' f^2 + f_y f f}{f_z}. \tag{5.5}$$

Combing this with Equation (5.1) we obtain an equivalent ODE for the DAE system. We differentiated the algebraic constraint twice, thus the DAE is of index-2. Differentiation is not generally used as a computational technique because properties of the original DAE, namely Equation (5.2), are often lost in numerical simulations of the differentiated equations [12].

5.2 What are Half-Explicit Methods

Half-explicit RK methods are explicit RK methods that enforce the algebraic constraint in an accurate way. For the numerical solution to a DAE of the form in Equations (5.1) and (5.2), we use the HERK method [10]

$$Y_i = y_{n-1} + h \sum_{j=1}^{i-1} a_{ij} f(Y_j, Z_j), \quad i = 1, ..., s, \tag{5.6}$$

$$0 = g(Y_i), \tag{5.7}$$

$$y_n = y_{n-1} + h \sum_{i=1}^{s} b_i f(Y_i, Z_i), \tag{5.8}$$

$$0 = g(y_n). \tag{5.9}$$

We also have the initial conditions y_0 and z_0 where $g(y_0) = 0$. The difference between RK and HERK is that we now have a Z_i component to our function f where we did not have one before. Also, we have a constraint function $g(Y_i)$. This system of equations is explicit for y', but it is implicit for $g(y)$, hence the name *half-explicit* RK [10]. Table 5.1 shows the Butcher Tableau of a particular HERK method of order three, which we call HERK3. We now work out the first few terms necessary to implement a third order, three stage HERK method where $i = 1, 2, 3$. We start by

0			
1/3	1/3		
1	-1	2	
	0	3/4	1/4

Table 5.1: The HERK3 method, from [10].

setting $Y_1 = y_0$ and note that $g(Y_1) = g(y_0) = 0$. Given

$$Y_2 = y_n + ha_{2,1}f(Y_1, Z_1), \tag{5.10}$$

$$g(Y_2) = 0 \tag{5.11}$$

we find Z_1 such that

$$g(y_n + ha_{2,1}f(Y_1, Z_1)) = 0. \tag{5.12}$$

Since Equation (5.12) is nonlinear, we use Newton's Method as our solver, which Brasley and Hairer [10] show converges if we use the initial guess $Z_1 = z_0$. Once we find Z_1, we use Equation (5.10) to find Y_2. We follow a similar procedure for Y_3 and Z_3, where

$$g(Y_3) = g(y_0 + h(a_{3,1}f(Y_1, Z_1) + a_{3,2}f(Y_2, Z_2))) = 0. \tag{5.13}$$

For the Newton solve, we have to take the derivative of $g(Y_i)$. For this derivative, we take the derivative with respect to Z_i. To alleviate some notation confusion, we define $G(Z_1) = g(Y_2)$ and $G(Z_2) = g(Y_3)$ and find the derivative with this notation. The general case for this derivative is

$$G_i'(Z_i) = g\left(y_0 + h\sum_{j=1}^{i-1} a_{ij}f(Y_j, Z_j)\right)(ha_{ij}f_{Z_i}(Y_i, Z_i)). \tag{5.14}$$

Now that we have determined all of the pieces required for our HERK methods, we present a low storage implementation.

5.2.1 Low Storage Implementation of HERK Methods

In Chapter 2, we gave Equation (2.10) as the low storage implementation of the standard RK method. At this point, we do a similar transformation of Equations (5.6)

and (5.9). The resulting equations yield

$$S_1^{[1]} = y_n, \tag{5.15}$$

$$\left.\begin{aligned} S_2^{[i+1]} &= A_i S_2^{[i]} + hf(S_1^{[i]}, Z_1^{[i]}), \\ S_1^{[i+1]} &= S_1^{[i]} + B_i S_2^{[i]}, \\ g(S_1^{[i+1]}) &= 0, \end{aligned}\right\} \quad i = 1, \ldots, s, \tag{5.16}$$

$$y_{n+1} = S_1^{[s]}. \tag{5.17}$$

We use

$$G_i(Z) = g\left(S_1 + B_i\left(A_i S_2 + hf\left(S_1, Z\right)\right)\right) = 0, \tag{5.18}$$

$$G_i'(Z) = g'\Big(S_1 + B_i\Big(A_i S_2 + hf\Big(S_1, Z\Big)\Big)\Big) B_i\Big(hf'\Big(S_1, Z\Big)\Big), \tag{5.19}$$

for $i = 1, \ldots, s$ together with the Newton's Method solver listed in Appendix F to find Z_1. We initialize the Newton solver with an initial guess for Z_1 from the previous time step.

5.2.2 Order Conditions

The order conditions for index-2 systems using HERK methods are formed in much the same way as we derived them in Chapter 2. We start with the Taylor expansion of the exact solution of (5.1). Next, we look at just one step of the method and rewrite (2.5) using

$$F_i(h) = f(Y_i, Z_i) \tag{5.20}$$

in (2.20). We take the first derivative of (2.20) and set $h = 0$ to get

$$z'(0) = \sum_{i=1}^{s} b_i F_i(0) = \sum_{i=1}^{s} b_i f(Y_i, Z_i). \tag{5.21}$$

This is associated with the first order condition as shown in Equation (2.26). We then take the derivative of Equation (5.21) and set $h = 0$ to find

$$z''(0) = 2\sum_{i=1}^{s} b_i F_i'(0) = \sum_{i=1}^{s} b_i (f_y Y_i' + f_z Z_i'). \tag{5.22}$$

For higher order methods, the additional derivates terms correspond to additional order conditions. For methods of order three, the order conditions that result are those already shown in Chapter 2 with two additional order conditions

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i \omega_{ij} c_{j+1}^2 = \frac{2}{3}, \tag{5.23}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i \omega_{ij} c_{j+1}^2 \omega_{ik} c_{k+1}^2 = \frac{4}{3}. \tag{5.24}$$

When solving for Z', we take an inverse. This results in the two new order conditions for third order half-explicit methods, where

$$\omega_{ij} = \begin{pmatrix} a_{2,1} & & & & \\ a_{3,1} & a_{3,2} & & & \\ \vdots & \vdots & \ddots & & \\ a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} & \\ b_1 & b_2 & \cdots & b_{s-1} & b_s \end{pmatrix}^{-1}. \tag{5.25}$$

Algebraically determining the order conditions is tedious. Therefore, we determine the order conditions from the rooted trees in much the same way as before. We represent the additional constraints by adding a fat node to the rooted tree that corresponds to ω. The remaining nodes in the rooted tree are known as meagre nodes. The two new half-explicit trees from Brasley and Hairer [10] are shown in Figure 5.1.

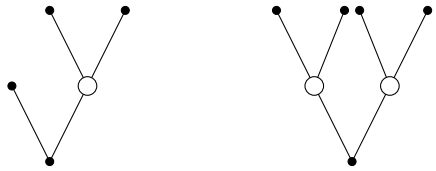


Figure 5.1: Third order half-explicit rooted trees.

To build the order conditions as we did with (2.44), we use ω_{ij} if the fat node j lies immediately above the meagre node i [10]. We add one to the index on each c_i that lies directly above a fat node. Note that we then define $c_{s+1} = 1$. We implement this in the code by artificially adding a one onto the end of the c vector [10]. As described in Hairer et al. [13] we are able to determine the order of each tree containing fat nodes by subtracting the number of fat nodes from the number of meagre nodes. Therefore, the two new trees are indeed third order rooted trees. We also needed the 14 order conditions for a fourth order method. The first eight fourth order conditions for HERK methods are

$$\sum_{i=1}^{s} b_i c_i^3 = \frac{1}{4}, \tag{5.26}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{8}, \tag{5.27}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j^2 = \frac{1}{12}, \tag{5.28}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{24}, \tag{5.29}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i^2 \omega_{ij} c_{j+1}^2 = \frac{1}{2}, \tag{5.30}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i c_i \omega_{ij} c_{j+1}^2 \omega_{ik} c_{k+1}^2 = 1, \tag{5.31}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{s} b_i\omega_{ij}c_{j+1}^2\omega_{ik}c_{k+1}^2\omega_{il}c_{l+1}^2 = 2, \tag{5.32}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i c_i\omega_{ij}c_{j+1}^3 = \frac{3}{4}. \tag{5.33}$$

For the six remaining order conditions, we need the relationship [10]

$$a_{s+1,i} = b_i, \quad i = 1,\dots,s. \tag{5.34}$$

Then we can write out the final six fourth order HERK order conditions

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} s b_i c_i\omega_{ij}c_{j+1}a_{j+1,k}c_k = \frac{3}{8}, \tag{5.35}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij}c_j\omega_{ij}c_{j+1}^2 = \frac{1}{4}, \tag{5.36}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i\omega_{ij}c_{j+1}^2\omega_{ik}c_{k+1}^3 = \frac{3}{2}, \tag{5.37}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{s} b_i\omega_{ij}c_{j+1}^2\omega_{ik}c_{k+1}^2 a_{k+1,l}c_l = \frac{3}{4}, \tag{5.38}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij}c_j\omega_{jk}c_{k+1}^2 = \frac{1}{6}, \tag{5.39}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{s} b_i a_{ij}\omega_{jk}c_{k+1}^2\omega_{jl}c_{l+1}^2 = \frac{1}{3}. \tag{5.40}$$

$$\tag{5.41}$$

These order conditions are used in Algorithm E.5.

5.3 Discovery and Optimization of LSHERK Methods

In order to find and optimize new LSHERK methods, we modified the code from Chapter 4. We focused on finding 14 stage methods, but now we change the order

to three due to the growing number of constraints. We incorporated the new order conditions into the equality constraint to ensure our method is third order. We now minimize a norm of the fourth order conditions. From the outset we chose to enforce the shape of the stability region using the ξ_i values from Table 5.2. As

i	ξ_i
1	1
2	1/2
3	1/3
4	1/6
5	2/3
6	4/3
7	1/24
8	$8.0971474827892589\times10^{-3}$
9	$1.2380169165300218\times10^{-3}$
10	$1.4920544370587013\times10^{-4}$
11	$1.4105197862197588\times10^{-5}$
12	$1.0338060754675449\times10^{-6}$
13	$5.7551620074656494\times10^{-8}$
14	$2.3518316167532871\times10^{-9}$
15	$6.6527970264862166\times10^{-11}$
16	$1.1639946786449694\times10^{-12}$
17	$9.4910013085549050\times10^{-15}$

Table 5.2: The shape constraints.

before, we bounded the coefficients to be between ±20.

To start the solver, we chose a random vector for x using a uniform distribution between ±2. However, the solver found no methods after numerous iterations. We then tried using the coefficients from NRK14C as a starting point for the initial guess. We perturbed a single coefficient for each optimization run by multiplying it by 0.9. The algorithm began returning usable results. However, this did not satisfying the order conditions to a high degree. Therefore, we tried another constrained nonlinear solver namely the SLSQP algorithm [14] from NLopt package [15]. We implemented the same constraints and used the coefficients returned by `fmincon` as the initial guess. Using this new solver, we were able to find coefficients that satisfied the order conditions to a high degree. The three best LSHERK methods we

found are listed in Tables 5.3 and 5.4.

OLSH14	
A_j	B_j
0	0.198689721501046
-1.536667718687070	0.092522019786294
-0.879071090817558	0.093317061742132
-0.336692132909897	0.015528392886136
-0.600875872026652	0.878595138450973
2.239571574791190	0.004847626739214
-0.922318030882955	0.489527826447674
0.429883109156452	0.738858611689567
-1.138227267743760	0.062964926879279
-2.645375146302570	0.018346991944929
0.208745274690937	0.323378610323469
-0.059235131779961	-0.023865839672009
-1.151123706135920	-0.294277657787366
-3.484587447077940	0.088153263710972

Table 5.3: The A_j and B_j coefficients for OLSH14.

5.4 Testing the DAE Solvers

We use Example 10.2 from Ascher and Petzold [8] as our DAE to test our new methods,

$$\begin{aligned} y_1' &= \left(10 - \frac{1}{2-t}\right) y_1 + (20 - 10t) z + \left(\frac{3-t}{2-t} e^t\right), \\ y_2' &= \left(\frac{-9}{t-2}\right) y_1 - y_2 - 9z + 2e^t, \\ 0 &= (t+2)\, y_1 + \left(t^2 - 4\right) y_2 - \left(t^2 + t - 2\right) e^t. \end{aligned} \tag{5.42}$$

This is an index-2 DAE. With the initial conditions $y_1(0) = y_2(0) = 1$, the exact solution is

$$y_1 = y_2 = e^t, \quad z = -\frac{e^t}{2-t}. \tag{5.43}$$

We solve the DAE using HERK3 and the three LSHERK methods in Tables 5.3 and 5.4. For HERK3 Algorithm F.1 serves as the time integrator function. We substituted the coefficients for each LSHERK method into Algorithm F.2.

O2LSH14		O3LSH14	
A_j	B_j	A_j	B_j
0	0.011606689535157	0	0.014228528646347
-0.74010204639345	0.193066237278654	-0.540982381029102	0.078478779562501
-2.54894597845811	2.05996751979073	-1.90680618151065	0.227585121053788
-13.7949081207664	0.212149307999438	-0.827743469943132	-1.49874337715584
-0.04878202484971	-0.080035183413218	-0.185077624731133	-0.00013227499321
-2.63597878427013	0.0692716851680793	-1.00299461809176	2.04238148887346
-10.8875767128353	0.0051616175210833	-2.17017565530989	0.000443666451349
0.04725137020767	0.0385851201335014	-0.989740743780935	2.78886431623001
1.57203594009144	-0.018839012403482	-2.62259296775679	5.57975124288274
-1.04849207573332	-0.024601846784907	-1.45037001707618	0.006220939198531
0.337171967245509	-2.28597480785815	-0.751383734763218	0.312734159646005
0.386611706418743	0.234175546664746	-0.782895050228605	0.137349425837006
0.474603255163912	0.203028289094254	0.285250619479765	1.01020457177164
-0.37187230970070	0.174243901777455	-14.1582458230965	0.049493455361853

Table 5.4: The coefficients for O2LSH14 and O3LSH14.

Figure 5.2 shows the solution error and convergence for each HERK method at the final time of $t = 1$. The best method was OLSH14 by a small margin, however, as h increases, O2LSH14 becomes the best method. We checked how well the numerical solution satisfies the constraint in Equation (5.9). Figure 5.3 shows the result of Equation (5.9) at the last time step of the HERK methods. All of the methods satisfy the constraint to near machine zero.

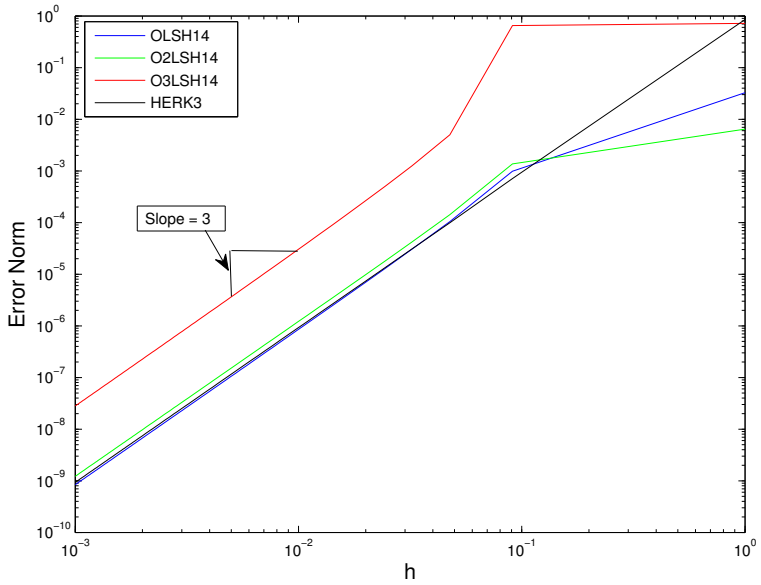


Figure 5.2: Error between the exact and numerical solutions. This plot also shows us that the methods have third order convergence.

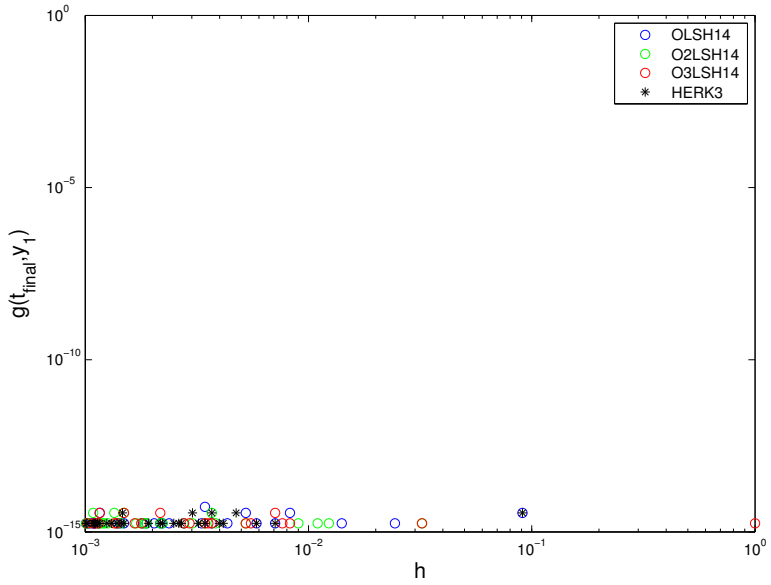


Figure 5.3: Plot of the DAE constraint $g(y) = 0$ evaluated at the solution y_1 for each method in this chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Conclusions and Future Work

For this thesis, we wanted to learn about and discover LSRK methods. To do this, we needed an understanding of the RK order conditions that make the methods consistent and accurate. Algebraically deriving these order conditions was difficult. For example, ninth order RK methods have 286 order conditions! Thus we explored generating the order conditions with rooted trees, which we found to be straightforward. After gaining a knowledge base of order conditions, we chose to evaluate the performance characteristics of RK4 compared with two LSRK methods, RK54 and NRK14C. For MOL PDE discretization of Maxwell's Equation, we found that NRK14C is more efficient when spatial error dominates. This validated the claim of Niegemann et al. [5] for when LSRK methods are more efficient. Niegemann et al. did not optimize the truncation error coefficients, which gave us the idea to do that. We discovered new LSRK methods, but the methods were not more efficient.

Reusing the tools from the optimization method, we looked at discovering methods for index-2 DAEs. First, we implemented and tested HERK3 as a baseline method for solving a DAE. We then ran the optimization method, which led to the discovery of three new LSHERK methods. Although the methods were accurate, their error levels were comparable to HERK3.

Areas for future work are:

- When optimizing for a new LSHERK method, different solvers could be used to produce better results. We stuck with `fmincon` for most of our work, but we did see a benefit using NLopt to condition the LSHERK coefficients further. Perhaps using a combination of solvers one could discover better performing LSHERK methods.
- In all of the optimization problems, determining the initial guess was a limiting factor. A more efficient method for exploring the high dimensional parameter

space would be beneficial.

- The next idea for the HERK problem would be to explore minimizing the TEC coefficients.
- The exploration of LSHERK for MOL discretization of PDEs.

APPENDIX A:
List of RK Order Conditions

Here is a list of the RK order conditions for first through fifth order. This list is in the order used throughout this work and used in the vector of γ values. The first through third order conditions are:

$$\sum_{i=1}^{s} b_i = 1 \tag{A.1}$$

$$\sum_{i=1}^{s} b_i c_i = \frac{1}{2} \tag{A.2}$$

$$\sum_{i=1}^{s} b_i c_i^2 = \frac{1}{3} \tag{A.3}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j = \frac{1}{6}. \tag{A.4}$$

The fourth order conditions are:

$$\sum_{i=1}^{s} b_i c_i^3 = \frac{1}{4} \tag{A.5}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{8} \tag{A.6}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j^2 = \frac{1}{12} \tag{A.7}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{24}. \tag{A.8}$$

The fifth order conditions are:

$$\sum_{i=1}^{s} b_i c_i^4 = \frac{1}{5} \tag{A.9}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i^2 a_{ij} c_j = \frac{1}{10} \tag{A.10}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i a_{ij} c_j^2 = \frac{1}{15} \tag{A.11}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i c_i a_{ij} a_{jk} c_k = \frac{1}{30} \tag{A.12}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} a_{ik} c_j c_k = \frac{1}{20} \tag{A.13}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j^3 = \frac{1}{20} \tag{A.14}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} c_j a_{jk} c_k = \frac{1}{40} \tag{A.15}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} a_{jk} c_k^2 = \frac{1}{60} \tag{A.16}$$

$$\sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{s} b_i a_{ij} a_{jk} a_{kl} c_l = \frac{1}{120}. \tag{A.17}$$

APPENDIX B:
Changes to the Maxwell's Equation in 2D Code

This appendix includes the two codes from Hesthaven and Warburton [9] that we changed to run our LSRK methods.

B.1 Time Integrator Function for Maxwell's Equation

This function runs the time integrator using the RK method of choice. We introduced the dtfactor here so we could iterate on different time step sizes.

```
function [Hx,Hy,Ez,time,dtscale,rhsevals] = Maxwell2D(Hx,Hy,Ez,FinalTime,dtfactor,RKmethod
    )

% function [Hx,Hy,Ez] = Maxwell2D(Hx, Hy, Ez, FinalTime)
% Purpose :Integrate TM-mode Maxwell's until FinalTime starting with
%          initial conditions Hx,Hy,Ez

Globals2D;
time = 0;

% Runge-Kutta residual storage
resHx = zeros(Np,K); resHy = zeros(Np,K); resEz = zeros(Np,K);

% compute time step size
rLGL = JacobiGQ(0,0,N); rmin = abs(rLGL(1)-rLGL(2));
dtscale = dtscale2D; %dt = min(dtscale)*rmin*2/3

%control dt size with dtfactor introduced below 30JAN15
%Matthew Fletcher thesis
dt = min(dtscale)*rmin*dtfactor;
% outer time step loop
rhsevals = 0;
while (time<FinalTime)

  if(time+dt>FinalTime), dt = FinalTime-time; end

   for INTRK = 1:length(RKmethod)
      % compute right hand side of TM-mode Maxwell's equations
      [rhsHx, rhsHy, rhsEz] = MaxwellRHS2D(Hx,Hy,Ez);
      rhsevals = rhsevals + 1;
      %rhsHx=0; rhsHy=0; rhsEz=0; %for eigenvalue part only
      % initiate and increment Runge-Kutta residuals
      resHx = RKmethod(INTRK,1)*resHx + dt*rhsHx;
      resHy = RKmethod(INTRK,1)*resHy + dt*rhsHy;
```

```
      resEz = RKmethod(INTRK,1)*resEz + dt*rhsEz;

      % update fields
      Hx = Hx+RKmethod(INTRK,2)*resHx; Hy = Hy+RKmethod(INTRK,2)*resHy;
      Ez = Ez+RKmethod(INTRK,2)*resEz;
   end;
   % Increment time
   time = time+dt;
end
return
```

Algorithm B.1: Function for solving Maxwell's Equation.

B.2 Driver File for Maxwell's Equation

This algorithm serves as the driver file for all of the codes associated with Maxwell's Equation in 2D. Here we iterated on the polynomial order. We also changed which RK method depending on the test at hand.

```
% Driver script for solving the 2D vacuum Maxwell's equations on TM form
Globals2D;
a=1;
b=1;
% Polynomial order used for approximation
tic
for k = [1 2]
    if k == 1
        RKmethod = NRK14C;
    elseif k == 2
        RKmethod = RK54;
    end
    for N = [4 6];% 8 10];

        % Read in Mesh
        [Nv, VX, VY, K, EToV] = MeshReaderGambit2D('Maxwell05.neu');

        % Initialize solver and construct grid and metric
        StartUp2D;

        % Set initial conditions
        mmode = 1; nmode = 1;
        icEz = sin(mmode*pi*x).*sin(nmode*pi*y); icHx = zeros(Np, K); icHy = zeros(Np, K);

        % Solve Problem
        FinalTime = 1;
        %RKmethod = NRK14C; %RK54; NRK14C; new14stage
        % compute time step size
```

```
        rLGL = JacobiGQ(0,0,N); rmin = abs(rLGL(1)-rLGL(2));
        dtscale = dtscale2D;
        % Exact Solution at FinalTime
        omega = pi*sqrt(mmode^2+nmode^2);
        exactHx = -pi*nmode/omega.* sin(mmode*pi*x).* cos(nmode*pi*y).* sin(omega*
    FinalTime);
        exactHy =  pi*mmode/omega.* cos(mmode*pi*x).* sin(nmode*pi*y).* sin(omega*
    FinalTime);
        exactEz =                  sin(mmode*pi*x).* sin(nmode*pi*y).* cos(omega*
    FinalTime);
        z=1;
        for dtfactor = 0.5:0.01:5%0.01:0.01:3%0.5:0.01:5
            [Hx,Hy,Ez,time,dtscale,rhsevals] = Maxwell2D(icHx,icHy,icEz,FinalTime,dtfactor
    ,RKmethod);

            ez_error = Ez - exactEz;
            MMez_error = MassMatrix*(J.*ez_error);
            l2ez_error = sqrt(ez_error(:)'*MMez_error(:));

            graph(z,a) = min(dtscale)*rmin*dtfactor;
            graph(z,a+1) = l2ez_error;
            opsct(z,b) = rhsevals;
            % line below needed if comparing ops count for one polynomial order
            % graph(z,a+2) = rhsevals;
            z=z+1;
        end
        a=a+2;
        b=b+1;
    end
end
toc

loglog(graph(:,1),graph(:,2),graph(:,3),graph(:,4),graph(:,5),graph(:,6),graph(:,7),graph
    (:,8))
title('dt vs. error','FontSize',14)
xlabel('dt','FontSize',14)
ylabel('error','FontSize',14)
legend('N=4; NRK14C','N=6; NRK14C','N=4; RK54','N=6; RK54')
```

Algorithm B.2: Driver script for solving the 2D vacuum Maxwell's Equation.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C:
Finding the Eigenvalues of the Discretization Operator for Maxwell's Equation

Our code uses L like this:

$$\frac{d}{dt}q_n = Lq_n, \text{ where } q_n = \begin{pmatrix} H_x \\ H_y \\ E_z \end{pmatrix}. \tag{C.1}$$

We need to write the code to return L in order to eventually find the eigenvalues. First, we remove the dtfactor loop from the original driver file and create a loop through n, where n is now the degrees of freedom per element (Np) multiplied by the number of elements (K) times three. The variables L and Q are then defined as a square matrix of size $Np \cdot K \cdot 3$ and 3D matrix Np by K by 3. We then solve `MaxwellRHS2D.m` using H_x, H_y, and E_z set equal to Q. Q is a matrix of all zeros except for the index of n, which is set to one. Therefore, as we loop through n, L is built from the output of `MaxwellRHS2D.m`. Once the algorithm is complete, we have the differentiation matrix, L. We then use Matlab's function `eig` to compute the eigenvalues of L. Taking the eigenvalues of L, we multiply by the time step size on the edge of stability for a specific method. To compute and plot the eigenvalues, we use the following code:

```
% Driver script for solving the 2D vacuum Maxwell's equations on TM form
Globals2D;
% Polynomial order used for approximation
N = 4;
for k = [1]
    if k == 1
        RKmethod = RK54;
    elseif k == 2
        RKmethod = NRK14C;
    end
    % Read in Mesh
    [Nv, VX, VY, K, EToV] = MeshReaderGambit2D('Maxwell05.neu');
    % Initialize solver and construct grid and metric
    StartUp2D;
```

```
    % Set initial conditions
    mmode = 1; nmode = 1;
    icEz = sin(mmode*pi*x).*sin(nmode*pi*y); icHx = zeros(Np, K); icHy = zeros(Np, K);
    FinalTime = 0;
    % compute time step size
    rLGL = JacobiGQ(0,0,N); rmin = abs(rLGL(1)-rLGL(2));
    dtscale = dtscale2D;
    L = zeros(Np*K*3);
    Q = zeros(Np, K, 3);
    for n = 1:Np*K*3
        Q(n) = 1;
        Hx = Q(:,:,1); Hy = Q(:,:,2); Ez = Q(:,:,3);
        [rhsHx, rhsHy, rhsEz] = MaxwellRHS2D(Hx,Hy,Ez);
        L(:,n) = [rhsHx(:);rhsHy(:);rhsEz(:)];
        Q(n) = 0;
    end
end
evs = eig(L); dt = 0.04125; scaledEV = dt*evs;
norm(real(scaledEV(real(scaledEV) > 0)), 'inf')
hold on
plot(scaledEV,'r*')
hold off
```

Algorithm C.1: Calculates and plots the eigenvalues of L.

APPENDIX D:
Optimization Algorithms for a New 14-Stage LSRK Method

This appendix covers the algorithms required to run `fmincon` to optimize a new 14 stage LSRK method.

D.1 Driver File for the Optimization

This algorithm functions as the driver file for the other algorithms in this appendix. We choose how we want to initialize the initial guess for `fmincon`. Included are three different options: NRK14C, zeros, or random numbers.

```
load('NRK14C.mat');
% initialize random number generator
rng(0,'twister');
% set upper and lower bounds for random numbers
var.m = -2; var.n =  0;  var.o = 0; var.p = 2;

options = optimset('diffmaxchange',Inf,'diffminchange',0, ...
                   'MaxFunEvals',600000,'TolX',0,'TolCon',1e-10,...
                   'TolFun',1e-10,'MaxIter',30000);
for i = 1
%     initial guess using a uniform distribution
    randA = (var.n-var.m).*rand(13,1)+var.m;
    randB = (var.p-var.o).*rand(14,1)+var.o;
%     Choose initial guess for x0 starting with NRK14C, all zeros or random numbers
%     [NRK14C(2:end,1);NRK14C(1:end,2)];%zeros(27,1);%[randA;randB]
    x0 = [randA;randB]
    [x,error] = condition_opt_driver(x0,options)
%     Check to see if new x satisfies the order conditions
    xA = [0;x(1:13)];
    xB = x(14:27);
    [a,b,c] = ConvertLSRK(xA,xB);
    for tol = [1e-13 1e-12 1e-10 1e-9]
        tol
        [satisfied,~] = OrderCondition(14,4,a,b,tol)
    end
end
```

Algorithm D.1: Driver algorithm for the optimization function.

D.2 Optimization Function

This algorithm sets the function we want to minimize, which in this case is the set of order conditions for fifth order methods.

```
function [ scalar ] = condition_opt_func( x0,var )

var.A_vec = [0;x0(1:13)];
var.B_vec = x0(14:27);
[A,b,c] = ConvertLSRK(var.A_vec,var.B_vec);
condition_vector5 = zeros(9,1);
for i = 1:var.s % single summation order conditions
    condition_vector5(1) = condition_vector5(1) + b(i)*(c(i)^4);
    for j= 1:var.s % double summation order conditions
        condition_vector5(2) = condition_vector5(2) + b(i)*(c(i)^2)*A(i,j)*c(j);
        condition_vector5(3) = condition_vector5(3) + b(i)*c(i)*A(i,j)*(c(j)^2);
        condition_vector5(6) = condition_vector5(6) + b(i)*A(i,j)*(c(j)^2);
        for k = 1:var.s % triple summation order conditions
            condition_vector5(4) = condition_vector5(4) + b(i)*c(i)*A(i,j)*A(j,k)*c(k);
            condition_vector5(5) = condition_vector5(5) + b(i)*A(i,j)*A(i,k)*c(j)*c(k);
            condition_vector5(7) = condition_vector5(7) + b(i)*A(i,j)*c(j)*A(j,k)*c(k);
            condition_vector5(8) = condition_vector5(8) + b(i)*A(i,j)*A(j,k)*(c(k)^2);
            for l = 1:var.s
                condition_vector5(9) = condition_vector5(9) + b(i)*A(i,j)*A(j,k)*A(k,l)*c(
    l);
            end
        end
    end
end
conditionsRHS = [1/5 1/10 1/15 1/30 1/20 1/20 1/40 1/60 1/120]';
scalar = norm(condition_vector5-conditionsRHS,2);
end
```

Algorithm D.2: OPtimization function for `fmincon`.

D.3 Options and Inputs Required to Run `fmincon`

Here we include any other constraints on the unknown variables. The upper and lower bounds limit the range of values `fmincon` can check to find a new LSRK method.

```
function [ cineq, ceq ] = condition_opt_constraints( x0, var )
%split x into A and B
var.A_vec = [0;x0(1:13)];
var.B_vec = x0(14:27);

[A,b,c] = ConvertLSRK(var.A_vec,var.B_vec);
```

```
condition_vector = zeros(18,1);
for i = 1:var.s % single summation order conditions
    condition_vector(1) = condition_vector(1) + b(i);
    condition_vector(2) = condition_vector(2) + b(i)*c(i);
    condition_vector(3) = condition_vector(3) + b(i)*(c(i)^2);
    condition_vector(5) = condition_vector(5) + b(i)*(c(i)^3);
    for j= 1:var.s % double summation order conditions
        condition_vector(4) = condition_vector(4) + b(i)*A(i,j)*c(j);
        condition_vector(6) = condition_vector(6) + b(i)*c(i)*A(i,j)*c(j);
        condition_vector(7) = condition_vector(7) + b(i)*A(i,j)*(c(j)^2);
        for k = 1:var.s % triple summation order conditions
            condition_vector(8) = condition_vector(8) + b(i)*A(i,j)*A(j,k)*c(k);
        end
    end
end
for  k = (var.p+1):var.s
  condition_vector(k+4) = condition_vector(k+4) + (b*A^(k-1)*ones(var.s,1));
end
conditionsRHS = [1 1/2 1/3 1/6 1/4 1/8 1/12 1/24 ...
    8.0971474827892589e-3  1.2380169165300218e-3 ...
    1.4920544370587013e-4  1.4105197862197588e-5 ...
    1.0338060754675449e-6  5.7551620074656494e-8 ...
    2.3518316167532871e-9  6.6527970264862166e-11 ...
    1.1639946786449694e-12 9.4910013085549050e-15]';
cineq = [];
ceq = abs(condition_vector-conditionsRHS);
end
```

Algorithm D.3: Sets the equality and inequality constraints for `fmincon`.

D.4 Optimization Constraints

The last algorithm contains the constraints on the first through fourth order conditions as well as the shape constraints.

```
function [x,error] = condition_opt_driver(x0, options)
% build vector for fmincon of initial conditions
% load LSRK method
load('NRK14C.mat');
var.s = length(NRK14C(:,:));
var.p = 4;

% fmincon constraints
A = []; B = []'; % inequality constraints
Aeq = []; Beq = []; % equality constraints
LB = [-20*ones(13,1);-20*ones(14,1)]';  % lower bound on the unknown variables
UB = [20*ones(13,1);20*ones(14,1)]';    % upper bound on the unknown variables

```

```
[x,error] = fmincon(@condition_opt_func,x0,A,B,Aeq,Beq,LB,UB, ...
                    @condition_opt_constraints,options,var)
```

Algorithm D.4: Sets the options for `fmincon`.

APPENDIX E:
Optimization Algorithms for a New Third Order LSHERK Method

This appendix covers the algorithms required to run `fmincon` to optimize a new 14 stage LSRK method.

E.1 Driver File for the Optimization

This algorithm functions as the driver file for the other algorithms in this appendix. We choose how we want to initialize the initial guess for `fmincon`. Included are three different options: NRK14C, zeros, or random numbers.

```
1
% initialize random number generator
3 rng(0,'twister');
% set upper and lower bounds for numbers
5
var.m = -2; var.n =  0; var.o = 0; var.p = 2;
7 options = optimset('diffmaxchange',Inf,'diffminchange',0, ...
        'MaxFunEvals',500000,'TolX',0,'TolCon',1e-10,...
9       'TolFun',1e-10,'MaxIter',30000);
for i = 1:10
11 %     initial guess using a uniform distribution
    randA = (var.n-var.m).*rand(13,1)+var.m;
13  randB = (var.p-var.o).*rand(14,1)+var.o;
%     Choose initial guess for x0 starting with NRK14C, all zeros or random numbers
15 %     [NRK14C(2:end,1);NRK14C(1:end,2)];%zeros(27,1);%[randA;randB]
    x0 = [randA;randB]
17  [x,error] = condition_opt_driver_half(x0,options)
%     Check to see if new x satisfies the half-explicit order conditions
19  xA = [0;x(1:13)];
    xB = x(14:27);
21  [a,b,c] = ConvertLSRK(xA,xB);
    for tol = [1e-7 1e-6]
23      tol
        [satisfied,~] = OrderCondition_HalfExplicit(14,3,a,b,tol)
25  end
end
```

Algorithm E.1: Driver algorithm for the optimization function.

E.2 Optimization Function

This algorithm sets the function we want to minimize, which in this case is the set of order conditions for fifth order methods.

```
function [ scalar ] = condition_opt_func_half( x0,var )

var.A_vec = [0;x0(1:13)];
var.B_vec = x0(14:27);
[A,b,c] = ConvertLSRK(var.A_vec,var.B_vec);
c = [c;1];
var.omega = inv([A(2:end,:);b]);
condition_vector5 = zeros(14,1);
for i = 1:var.s % single summation order conditions
    condition_vector5(1) = condition_vector5(1) + b(i)*(c(i)^3);
    for j= 1:var.s % double summation order conditions
        condition_vector5(2)  = condition_vector5(2) + b(i)*c(i)*A(i,j)*c(j);
        condition_vector5(3)  = condition_vector5(3) + b(i)*A(i,j)*(c(j)^2);
        condition_vector5(5) = condition_vector5(5) + b(i)*(c(i)^2)*var.omega(i,j)*(c(j+1)
    ^2);
        condition_vector5(8) = condition_vector5(8) + b(i)*c(i)*var.omega(i,j)*(c(j+1)^3);
        condition_vector5(10) = condition_vector5(10) + b(i)*A(i,j)*c(j)*var.omega(i,j)*(c
    (j+1)^2);
        for k = 1:var.s % triple summation order conditions
            condition_vector5(4) = condition_vector5(4) + b(i)*A(i,j)*A(j,k)*c(k);
            condition_vector5(6) = condition_vector5(6) + b(i)*c(i)*var.omega(i,j)*(c(j+1)
    ^2)*var.omega(i,k)*(c(k+1)^2);
            if j == var.s
                condition_vector5(9) = condition_vector5(9) + b(i)*c(i)*var.omega(i,j)*c(j
    +1)*b(k)*c(k);
            else
                condition_vector5(9) = condition_vector5(9) + b(i)*c(i)*var.omega(i,j)*c(j
    +1)*A(j+1,k)*c(k);
            end
            condition_vector5(11) = condition_vector5(11) + b(i)*var.omega(i,j)*(c(j+1)^2)
    *var.omega(i,k)*(c(k+1)^3);
            condition_vector5(13) = condition_vector5(13) + b(i)*A(i,j)*c(j)*var.omega(j,k
    )*(c(k+1)^2);
            for l = 1:var.s % quadruple summation order conditions
                condition_vector5(7) = condition_vector5(7) + b(i)*var.omega(i,j)*(c(j+1)
    ^2)*var.omega(i,k)*(c(k+1)^2)*var.omega(i,l)*(c(l+1)^2);
                if k == var.s
                    condition_vector5(12) = condition_vector5(12) + b(i)*var.omega(i,j)*(c
    (j+1)^2)*var.omega(i,k)*(c(k+1)^2)*b(l)*c(l);
                else
                    condition_vector5(12) = condition_vector5(12) + b(i)*var.omega(i,j)*(c
    (j+1)^2)*var.omega(i,k)*(c(k+1)^2)*A(k+1,l)*c(l);
                end
                condition_vector5(14) = condition_vector5(14) + b(i)*A(i,j)*var.omega(j,k)
```

```
    *(c(k+1)^2)*var.omega(j,l)*(c(l+1)^2);
            end
        end
    end
end
conditionsRHS = [1/4 1/8 1/12 1/24 1/2 1 2 3/4 3/8 1/4 3/2 3/4 1/6 1/3]';
scalar = norm(condition_vector5-conditionsRHS,2);
end
```

Algorithm E.2: Sets the optimization function according to the 3^{rd} and 4^{th} order half-explicit RK coefficients.

E.3 Options and Inputs Required to Run `fmincon`

Here we include any other constraints on the unknown variables. The upper and lower bounds limit the range of values `fmincon` can check to find a new LSRK method.

```
function [ cineq, ceq ] = condition_opt_constraints_half( x0,var )

%split x into A and B
var.A_vec = [0;x0(1:13)];
var.B_vec = x0(14:27);
% Constraints for the Half-Explicit coefficients and circular shape
conditionsRHS = [1 1/2 1/3 1/6 2/3 4/3 1/24 ...
    8.0971474827892589e-3  1.2380169165300218e-3 ...
    1.4920544370587013e-4  1.4105197862197588e-5 ...
    1.0338060754675449e-6  5.7551620074656494e-8 ...
    2.3518316167532871e-9  6.6527970264862166e-11 ...
    1.1639946786449694e-12 9.4910013085549050e-15]';
[A,b,c] = ConvertLSRK(var.A_vec,var.B_vec);
c = [c;1];
var.omega = inv([A(2:end,:);b]);
condition_vector = zeros(length(conditionsRHS),1);
for i = 1:var.s % single summation order conditions
    condition_vector(1) = condition_vector(1) + b(i);
    condition_vector(2) = condition_vector(2) + b(i)*c(i);
    condition_vector(3) = condition_vector(3) + b(i)*(c(i)^2);
    for j= 1:var.s % double summation order conditions
        condition_vector(4) = condition_vector(4) + b(i)*A(i,j)*c(j);
        condition_vector(5) = condition_vector(5) + b(i)*c(i)*var.omega(i,j)*(c(j+1)^2);
        for k = 1:var.s % triple summation order conditions
            condition_vector(6) = condition_vector(6) + b(i)*var.omega(i,j)*(c(j+1)^2)*var
    .omega(i,k)*(c(k+1)^2);
        end
    end
end
```

```
for  l = (var.p+1):var.s
  condition_vector(l+3) = condition_vector(l+3) + (b*A^(l-1)*ones(var.s,1));
end

cineq = [];
ceq = abs(condition_vector-conditionsRHS);

end
```

Algorithm E.3: Sets the constraints for 3rd order half-explicit RK coefficients as well as the shape constraints.

E.4 Optimization Constraints

The last algorithm contains the constraints on the first through fourth order conditions as well as the shape constraints.

```
function [x,error] = condition_opt_driver_half(x0, options)

var.s = 14;% length(NRK14C(:,:));
var.p = 3;
% fmincon constraints

A = []; B = []'; % inequality constraints
Aeq = []; Beq = []; % equality constraints
LB = [-20*ones(13,1);-20*ones(14,1)]';  % lower bound on the unknown variables
UB = [20*ones(13,1);20*ones(14,1)]'; % upper bound on the unknown variables

[x,error] = fmincon(@condition_opt_func_half,x0,A,B,Aeq,Beq,LB,UB, ...
                    @condition_opt_constraints_half,options,var)
```

Algorithm E.4: Runs the `fmincon` optimization function.

E.5 Check Order Conditions and Truncation Error Coefficient

This algorithm is similar to the previous algorithms where we check all of the order conditions for a given RK method. Here we use up to the fourth order half-explicit RK order conditions. The last few lines comprise the truncation error coefficient calculation.

```
function [satisfied,c,upsilon] = TEC_HalfExplicit(s,p,A,b,tol)
% s = number of stages
```

```
% p = order of method
% A = tableau
% b = weights of A
% tol = tolerance
if p == 3
    Tc = 6;
elseif p == 4
    Tc = 20;
end
c = sum(A,2);
Cmatrix = diag(c);
one = ones(s,1);
satisfied = true;
omega = inv([A(2:end,:);b]);
% add on 1 to c for the j+1 and k+1 index from (3.3) in Brasey and Hairer
c = [c;1];
for l = 1:p
    for k = 0:(p-l)
        cond = b * A^k * Cmatrix^(l-1) * one;
        ans2 = factorial(l-1)/factorial(k+l);
        %introduced tolerance condition
        if abs(cond-ans2) > tol
            satisfied = false;
        end
    end
end
conditionsRHS = [1 1/2 1/3 1/6 2/3 4/3 1/4 1/8 1/12 1/24 1/2 1 2 3/4 3/8 1/4 3/2 3/4 1/6
    1/3];
condition_vector = zeros(1,length(conditionsRHS));
for i = 1:s % single summation order conditions
    condition_vector(1) = condition_vector(1) + b(i);
    condition_vector(2) = condition_vector(2) + b(i)*c(i);
    condition_vector(3) = condition_vector(3) + b(i)*(c(i)^2);
    condition_vector(7) = condition_vector(7) + b(i)*(c(i)^3);
    for j= 1:s % double summation order conditions
        condition_vector(4)  = condition_vector(4) + b(i)*A(i,j)*c(j);
        condition_vector(5)  = condition_vector(5) + b(i)*c(i)*omega(i,j)*(c(j+1)^2);
        condition_vector(8)  = condition_vector(8) + b(i)*c(i)*A(i,j)*c(j);
        condition_vector(9)  = condition_vector(9) + b(i)*A(i,j)*(c(j)^2);
        condition_vector(11) = condition_vector(11) + b(i)*(c(i)^2)*omega(i,j)*(c(j+1)^2);
        condition_vector(14) = condition_vector(14) + b(i)*c(i)*omega(i,j)*(c(j+1)^3);
        condition_vector(16) = condition_vector(16) + b(i)*A(i,j)*c(j)*omega(i,j)*(c(j+1)
    ^2);
        for k = 1:s % triple summation order conditions
            condition_vector(6)  = condition_vector(6) + b(i)*omega(i,j)*(c(j+1)^2)*omega(
    i,k)*(c(k+1)^2);
            condition_vector(10) = condition_vector(10) + b(i)*A(i,j)*A(j,k)*c(k);
            condition_vector(12) = condition_vector(12) + b(i)*c(i)*omega(i,j)*(c(j+1)^2)*
    omega(i,k)*(c(k+1)^2);
```

```
            if j == s
                condition_vector(15) = condition_vector(15) + b(i)*c(i)*omega(i,j)*c(j+1)*b(k)*c(k);
            else
                condition_vector(15) = condition_vector(15) + b(i)*c(i)*omega(i,j)*c(j+1)*A(j+1,k)*c(k);
            end
            condition_vector(17) = condition_vector(17) + b(i)*omega(i,j)*(c(j+1)^2)*omega(i,k)*(c(k+1)^3);
            condition_vector(19) = condition_vector(19) + b(i)*A(i,j)*c(j)*omega(j,k)*(c(k+1)^2);
            for l = 1:s % quadruple summation order conditions
                condition_vector(13) = condition_vector(13) + b(i)*omega(i,j)*(c(j+1)^2)*omega(i,k)*(c(k+1)^2)*omega(i,l)*(c(l+1)^2);
                if k == s
                    condition_vector(18) = condition_vector(18) + b(i)*omega(i,j)*(c(j+1)^2)*omega(i,k)*(c(k+1)^2)*b(l)*c(l);
                else
                    condition_vector(18) = condition_vector(18) + b(i)*omega(i,j)*(c(j+1)^2)*omega(i,k)*(c(k+1)^2)*A(k+1,l)*c(l);
                end
                condition_vector(20) = condition_vector(20) + b(i)*A(i,j)*omega(j,k)*(c(k+1)^2)*omega(j,l)*(c(l+1)^2);
            end
        end
    end
end

for z = 1:length(condition_vector)
    if abs(condition_vector(z)-conditionsRHS(z)) <= tol
        success = ['Passes order condition ',num2str(z),'.'];
        display(success)
        %satisfied = 0;
    end
end
c = sum(A,2);
% Truncation Error Coefficient Calculation
upsilon = zeros(1,Tc);
phi = condition_vector(1,1:Tc); gamma = conditionsRHS(1,1:Tc);
alpha = [1 1 1 1 1 1]; rho = [1 2 3 3 3 3];
upsilon = (phi.*gamma-1).*(alpha./factorial(rho));
```

Algorithm E.5: Checks the order conditions for a given half-explicit RK method and outputs the TEC.

APPENDIX F:
Algorithms for Solving a DAE

This appendix covers the algorithms required to solve the DAE using a Newton's solve in the integrator for both HERK3 and the LSHERK methods.

F.1 HERK3 Method

This algorithm functions as the driver file for the other algorithms in this appendix. We choose how we want to initialize the initial guess for `fmincon`. Included are three different options: NRK14C, zeros, or random numbers.

```
function [y1,z1,t1] = HERK3(y0,z0,f,g,df,dg,t0,h,n)
% t0, y0 and z0 are the initial conditions
% f and g are anonymous function from the DAE
% df is the derivative of f with respect to z
% dg is the derivative of g with respect to y
% h is the step size
% n is the number of iterations

a = [0 0 0;1/3 0 0;-1 2 0];
b = [0 3/4 1/4];
c = sum(a,2);
for k  = 1:n
    tol = 1e-7;
    Y1  = y0;
    t1  = t0;
    t2  = t0 + h*c(2);
    t3  = t0 + h*c(3);

    G1     = @(z) g(t2,y0 + h * a(2,1) * f(t1,Y1,z));
    dG1    = @(z) h * a(2,1) * (dg(t2,y0 + h * a(2,1) * f(t1,Y1,z)) * df(t1,Y1,z));
    [Z1,~] = newton(G1,dG1,z0,tol,10001,0);

    Y2     = y0 + h * a(2,1) * f(t1,Y1,Z1);
    G2     = @(z) g(t3,y0 + h * (a(3,1) * f(t1,Y1,Z1) + a(3,2) * f(t2,Y2,z)));
    dG2    = @(z) h * a(3,2) * (dg(t3,y0 + h * (a(3,1) * f(t1,Y1,Z1) + a(3,2) * f(t2,Y2,z)
    )) * df(t2,Y2,z));
    [Z2,~] = newton(G2,dG2,z0,tol,10001,0);

    Y3     = y0 + h * (a(3,1) * f(t1,Y1,Z1) + a(3,2) * f(t2,Y2,Z2));
    G3     = @(z) g(t0 + h,y0 + h *(b(2) * f(t2,Y2,Z2) + b(3) * f(t3,Y3,z)));
    dG3    = @(z) h * b(3) * (dg(t0 + h,y0 + h *(b(2) * f(t2,Y2,Z2) + b(3) * f(t3,Y3,z)))*
    df(t3,Y3,z));
```

```
    [Z3,~] = newton(G3,dG3,z0,tol,10001,0);

    y1 = y0 + h * (b(2) * f(t2,Y2,Z2) + b(3) * f(t3,Y3,Z3));
    z1 = Z3;
    t1 = t0 + h;
    y0 = y1;
    z0 = z1;
    t0 = t1;
end
end
```

Algorithm F.1: This function evaluates a function using the HERK3 method, from [10].

F.2 LSRK Method

This algorithm sets the function we want to minimize, which in this case is the set of order conditions for fifth order methods.

```
function [S1,t0] = LSHERK(f,y0,z0,g,df,dg,A,B,c,t0,h,n,tol)
% t0, y0 and z0 are the initial conditions
% f and g are anonymous function from the DAE
% df is the derivative of f with respect to z
% dg is the derivative of g with respect to y
% h is the step size
% n is the number of iterations
% A, B, and c are the LSRK coefficients
% tol is the tolerance for the Newton solve
s = length(A);
c = [c;1];
S1 = y0;
S2 = zeros(size(y0));
Z1 = z0;
for i = 1:n
    for j = 1:s

        G  = @(z)   g(t0+c(j+1)*h,S1+B(j)*(A(j)*S2+h.*f(t0+c(j)*h,S1,z)));
        dG  = @(z) dg(t0+c(j+1)*h,S1+B(j)*(A(j)*S2+h.*f(t0+c(j)*h,S1,z)))*B(j)*(h.*df(t0+c
    (j)*h,S1,z));
        [Z1,~] = newton(G,dG,z0,tol,10001,0);

        S2 = A(j)*S2+h.*f(t0+c(j)*h,S1,Z1);
        S1 = S1+B(j)*S2;

    end
    t0 = t0 + h;
    z0 = Z1;
end
end
```

Algorithm F.2: This function takes evaluates a function using the LSHERK method.

F.3 Newton's Solver

The last algorithm contains the constraints on the first through fourth order conditions as well as the shape constraints.

```
1
% NEWTON Newton's method for finding the roots of a scalar equation
3 % inputs:
%   f       function handle for the function f(x)
5 %   fp      function handle for the derivative function f'(x)
%   x0      initial guess for the root
7 %   tol     absolute tolerance
%   N       maximum number of iterations
9 %   output  a boolean that if true causes the program to display the number
%           of the iterations, the absolute convergence criterion, and the
11 %           function value, i.e.,
%                disp([n,abs(xk-xk1),f(xk)]);
13 % outputs:
%   X     found value for the root
15 %   n     number of iterations (return value of N+1 indicates failure to
%         find the root in N iterations)
17
function [X,n] = newton(f,fp,x0,tol,n,output)
19 loop = 0;
if output
21     disp('  Iterations   Abs Error   Value  ')
end
23 for k=1:n
    x1=x0-(f(x0)/fp(x0));
25     % Algorithm from "A First Course in Numerical Methods" by Ascher and Grief;
    % Chapter 3.4 page 51.
27     if output
        disp([k,abs(x1-x0),f(x0)]);
29     end
    if abs(x1-x0)<tol
31         n=k;
        X=x1;
33         break;
    elseif (k == n)
35         error( 'Newtons method did not converge' );
    end
37
    x0=x1;
39 end
```

Algorithm F.3: This algorithm is the Newton's Method we used to solve for the Z_i of our function.

F.4 Driver File for Solving the DAE

This algorithm is similar to the previous algorithms where we check all of the order conditions for a given RK method. Here we use up to the fourth order half-explicit RK order conditions. The last few lines comprise the truncation error coefficient calculation.

```
1
% Parameters
3 a = 10;
t0 = 0;
5 Yp = @(t) [(a-(1/(2-t))) 0;(1-a)/(t-2) -1];
Zp = @(t) [(2-t)*a;a-1];
7 f  = @(t,y,z) Yp(t)*y+Zp(t)*z+[(3-t)/(2-t)*exp(t);2*exp(t)];

9 g  = @(t,y) [(t+2) (t^2-4)]*y-(t^2+t-2)*exp(t);
df = @(t,y,z) Zp(t);
11 dg = @(t,y) [t+2 t^2-4];

13 y0 = [1; 1];
z0 = -exp(t0)/(2-t0);
15 tf = 1;

17 %load LSRK coefficients
load('OHERK14.mat')
19 [~,~,c1] = ConvertLSRK(OHERK14(:,1),OHERK14(:,2));
A1 = OHERK14(:,1);
21 B1 = OHERK14(:,2);

23 q = 1;

25 for n = 1:10:1001
    h  = (tf-t0)/n;
27     err1(1,q) = h;
    [y1,z1,t1] = HERK3(y0,z0,f,g,df,dg,t0,h,n);
29     %[S1,t] = LSHERK(f,y0,z0,g,df,dg,A1,B1,c1,t0,h,n,1e-7);
    yexact = [exp(tf);exp(tf)];
31     err1(2,q) = norm((y1-yexact),inf);
    err1(3,q) = g(tf,y1);
33     q=q+1;
end
35
hold on
```

```
figure(1)
loglog(err1(1,:),err1(2,:),'r-')
title('Error','FontSize',14)
xlabel('dt','FontSize',14)
ylabel('Error Norm','FontSize',14)
legend('OHERK14','O2HERK14','O3HERK14','HERK3')

figure(2)
loglog(err1(1,:),err1(3,:),'bo')
title('Error','FontSize',14)
xlabel('dt','FontSize',14)
ylabel('g(t_{final},y_{1})','FontSize',14)
legend('OHERK14','O2HERK14','O3HERK14','HERK3')
```

Algorithm F.4: This algorithm sets up the DAE and solves it using either a LSHERK method or HERK3.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] J. C. Butcher, *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. New York, NY, USA: Wiley-Interscience, 1987.

[2] J. H. Williamson, "Low-storage Runge-Kutta schemes," *Journal of Computational Physics*, vol. 35, no. 1, pp. 48–56, Mar 1980.

[3] D. I. Ketcheson, "Runge-kutta methods with minimum storage implementations," *Journal of Computational Physics*, vol. 229, no. 5, pp. 1763 – 1773, 2010.

[4] M. H. Carpenter and C. A. Kennedy, "Fourth-order 2N-storage Runge-Kutta schemes," *NASA TM*, vol. 109112, 1994.

[5] J. Niegemann, R. Diehl, and K. Busch, "Efficient low-storage Runge-Kutta schemes with optimized stability regions," *Journal of Computational Physics*, vol. 231, no. 2, pp. 364 – 372, 2012.

[6] F. Cameron, "A Matlab package for automatically generating Runge-Kutta trees, order conditions, and truncation error coefficients," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 2, pp. 274–298, 2006.

[7] P. Bogacki and S. LF, "A 3(2) pair of Runge-Kutta formulas," *Applied Mathematical Letters*, vol. 2, no. 4, pp. 321–325, 1989.

[8] U. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, 1st ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1998.

[9] J. S. Hesthaven and T. Warburton, *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. New York, NY, USA: Springer, 2008, vol. 54.

[10] V. Brasey and E. Hairer, "Half-explicit Runge-Kutta methods for differential-algebraic systems of index 2," *SIAM Journal on Numerical Analysis*, vol. 30, no. 2, pp. 538–552, 1993.

[11] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. New York, NY, USA: SIAM, 1996, vol. 14.

[12] Wolfram|Alpha. (2009). Numerical solution of differential-algebraic equations. [Online]. Available: http://www.wolframalpha.com/input/?i=2%2B2. Accessed: 2015-05-31.

[13] E. Hairer, M. Roche, and C. Lubich, *The numerical solution of differential-algebraic systems by Runge-Kutta methods*, ser. Lecture Notes in Mathematics. Berlin: Springer, 1989.

[14] D. Kraft, "Algorithm 733: TOMP8211;fortran modules for optimal control calculations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 20, no. 3, pp. 262–281, Sep 1994. [Online]. Available: http://doi.acm.org/10.1145/192115.192124

[15] S. G. Johnson. (2010). The NLopt nonlinear-optimization package. [Online]. Available: http://ab-initio.mit.edu/nlopt. Accessed: 31 May, 2015.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California